TOPOLOGY OPTIMIZATION WITH UNSTRUCTURED MESHES ON GRAPHICS PROCESSING UNITS (GPUS)

BY

TOMÁS ZEGARD LATRACH

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Civil Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2010

Urbana, Illinois

Adviser:

Professor Glaucio H. Paulino

Abstract

The present work investigates the feasibility of finite element methods and topology optimization for unstructured meshes in massively parallel computer architectures, more specifically on Graphics Processing Units or GPUs. Algorithms for every step in these methods are proposed and benchmarked with varied results. The ultimate goal of this work is to speed up the topology optimization process by means of parallel computing using off-the-shelf hardware. To further facilitate future application and deployment, a transparent massively parallel topology optimization code was written and tested. Examples are compared with both, a standard sequential version of the code, and a massively parallel version to better illustrate the advantages and disadvantages of this approach.

To my mom, Ketty *and my dad,* Gastón.

Acknowledgments

I would like to show my gratitude to all the people that gave me their support during the development of the present work.

I owe my deepest gratitude to my parents, who have always unconditionally encouraged and supported me in every endeavor I take. I would like to thank my advisor Glaucio H. Paulino for his help, encouragement and excitement, that ultimately led to the present work. I would also like to thank my colleagues Marco Alfano, Daiane Brisotto, Youn-Sha Chan, Rodrigo Espinha, Wenbo Fang, Arun Gain, Sofie Leon, Tam Nguyen, Kyoungsoo Park, Daniel Spring, Lauren Stromberg, Alok Sutradhar, Cameron Talischi and Ying Yu, for their help, animosity and encouragement. They all are key contributors to the fun and exciting work environment I had during the development of this work. I want to thank as well Eric de Sturler for his comments, help and suggestions.

Additionally, I am grateful to the Fulbright Program and the department of Civil Engineering of the University of Illinois at Urbana-Champaign for believing in me, and for the opportunity to become a better person in every sense.

Finally, I would like to thank all of my family and friends in Chile. Even thought they are pretty far in distance, they all give me close support.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

API	Application Programming Interface.
BFS	Breadth-First Search.
BIP	Binary Integer Programming.
CAMD	Continuous Approximation of Material Distribution.
CAS	Computer Algebra System.
CPU	Central Processing Unit.
CUDA	NVIDIA's Compute Unified Device Architecture.
CST	Constant Strain Triangle or Constant Stress Triangle.
CTM	ATI's Close-to-the-Metal.
DFS	Depth-First Search.
DOF	Degree Of Freedom.
FEA	Finite Element Analysis.
FEM	Finite Element Method.
FDM	Finite Difference Method.
FIFO	First In, First Out.
FLOPS	Floating Point Operations Per Second.
GPGPU	General-Purpose Computing on Graphics Processing Units.
GPU	Graphics Processing Unit.
LIFO	Last In, First Out.
MBB	Messerschmitt-Bölkow-Blohm.
MMA	Method of Moving Asymptotes.
MTOP	Multi-resolution Topology Optimization.
NaN	Not a Number.
OC	Optimality Criteria.
SLP	Sequential Linear Programming.

SZEM	Spurious Zero-Energy Mode.
PC	Personal Computer.
PDE	Partial Differential Equation.
SIMP	Solid Isotropic Material with Penalization.
TOP	Topology Optimization.

Symbols

$[B]$	Strain-Displacement matrix.
b_w	Bandwidth of a matrix.
c	Compliance $c = \{f\} \cdot \{u\}$.
C_G	Centroid or geometric center.
C_M	Center of mass.
$\{D\}$	Imposed displacements vector.
$[D]$	Constitutive matrix.
E	Young's modulus.
f	Volume fraction.
$\{F\}$	Global force vector.
$\{f\}$	Local force vector.
$\hat{H}_f$	Convolution function.
J	Determinant of the transformation Jacobian $[J]$.
$[J]$	Transformation Jacobian.
$[K]$	Global stiffness matrix.
$[k]$	Local stiffness matrix.
$[L]$	Lower triangular matrix.
$L\left(G^C\right)$	Communication matrix for graph G.
m	Density move limit.
N	Shape function.
n	Number of elements, matrix size or other depending on the context.
p	Penalization factor for SIMP.
q	Nodal force.
$\{R\}$	Reaction vector (forces).
r_{min}	Filter radius.

$[U]$	Upper triangular matrix.
V	Volume.
w	Gauss point weight.
$\{q\}$	Internal force vector.
u	Displacement.
Γ	Boundary of domain Ω.
γ	Shear strain.
ε	Normal strain.
ε_0	Initial strain.
λ	Lagrange multiplier.
ν	Poisson's ratio.
Ω	Domain.
ρ	Density.
σ	Normal stress.
σ_0	Initial stress.
τ	Shear stress.
ξ	Gauss point location.
$\chi(G)$	Chromatic number for graph G.

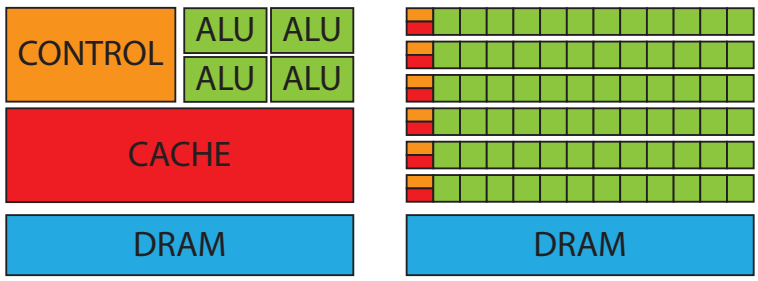
Chapter 1

Introduction

Computers have been developing at an amazing speed ever since their introduction. Moore's Law has successfully predicted exponential increase of processing speed for many decades [31]. Until very recently, the same software ran faster with each new processor. The processor speed kept continuously increasing without any mayor architectural changes until approximately 2003, when power consumption and other problems put a limit on the processor's clock. Since then, processors kept getting faster by increasing parallelism [43].

Many-core processors are a type of processors that evolved to a very high level of parallelism. A class of many-core processors are the *Graphics Processing Units* or GPUs for short. Practically all personal computers (PCs) have GPUs in them. Both GPUs and central processing units (CPUs) have two very different design approaches (Figure (1.1)): The CPU is a general purpose multi-core processor, with many and very high level instructions, whereas the GPU is a many-core processor with very fast and smaller set of instructions (more specialized type of hardware). Three architectures follow this many-core trend: Cell processor (developed by IBM, Sony and Toshiba), the NVIDIA GPUs (developed by NVIDIA) and the ATI GPUs (developed by AMD through ATI).

The GPUs were not originally designed to be used as a compute co-processor, but for graphics. Because of



(a) Schematic of a CPU (b) Schematic of a GPU

Figure 1.1: Comparison of the basic diagram of a CPU and a GPU.

that, early numerical computing attempts had to use the graphics application programming interface (API). Using the graphics API for numerical computing is extremely complex and has little flexibility due to the bounds imposed by the graphics API. Nevertheless these early attempts showed the tremendous amount of computing power available in the GPU (Figure (1.2)). This approach is called *General-Purpose Computing on Graphics Processing Units* or GPGPU, which never gained to much popularity because of its difficulty, and was considered a rather obscure technique [20, 46, 35]. In 2007 NVIDIA released the *Compute Unified Device Architecture* or CUDA for short [80]. It allowed programmers to use NVIDIA GPUs to develop massively parallel computing applications in an easy way [19]. There are other attempts to do this, but none of them are as popular and widespread as CUDA. Some other implementations, similar to CUDA are: ATI's Close-to-the-Metal (CTM), ATI's Stream and OpenCL. One of the main advantages of CUDA, is that it is an extension to C++, that eases the learning curve for programmers [65].

In order to take advantage of the high level of parallelism in these many-core processors, software must be made so that several tasks can be concurrently executed without interfering with each other. Code previously made for single or few-core CPUs cannot be directly ported to a many-core architecture and, in most cases, the code structure must be completely changed or re-thinked to make it work efficiently [62, 63, 56].

Topology optimization is a technique that seeks to optimally distribute the material in a domain in order

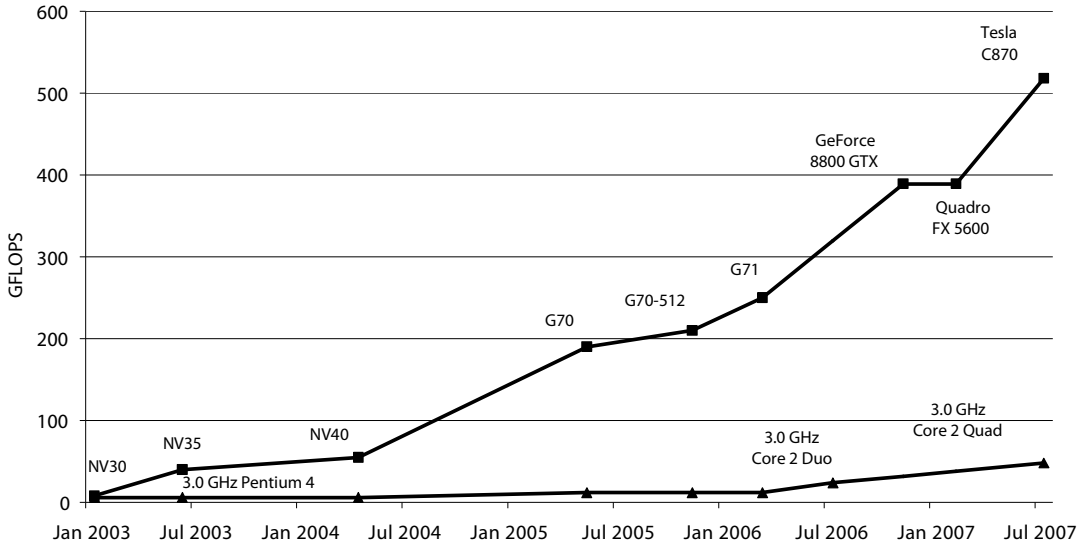


Figure 1.2: Theoretical billions of floating point operations (GFLOPS) peak of NVIDIA GPUs and typical CPUs (extracted from [61]).

to obtain an *ideal structure* [54]. Material distribution automatically generates structural members, holes and other topologies as needed. The downside of the technique is the tremendous amount of computation

required. That is why this work is focused in speeding it up using many-core processors, specifically, GPUs. The topology optimization code developed for this work has 2 user selectable compute chains, both with a solver variation posibility, resulting in a total of 4 possible compute chains as depicted in Figure (1.3). The user can choose for the topology optimization loop to take place entirely in the GPU, entirely in the CPU, or mixtures of both.

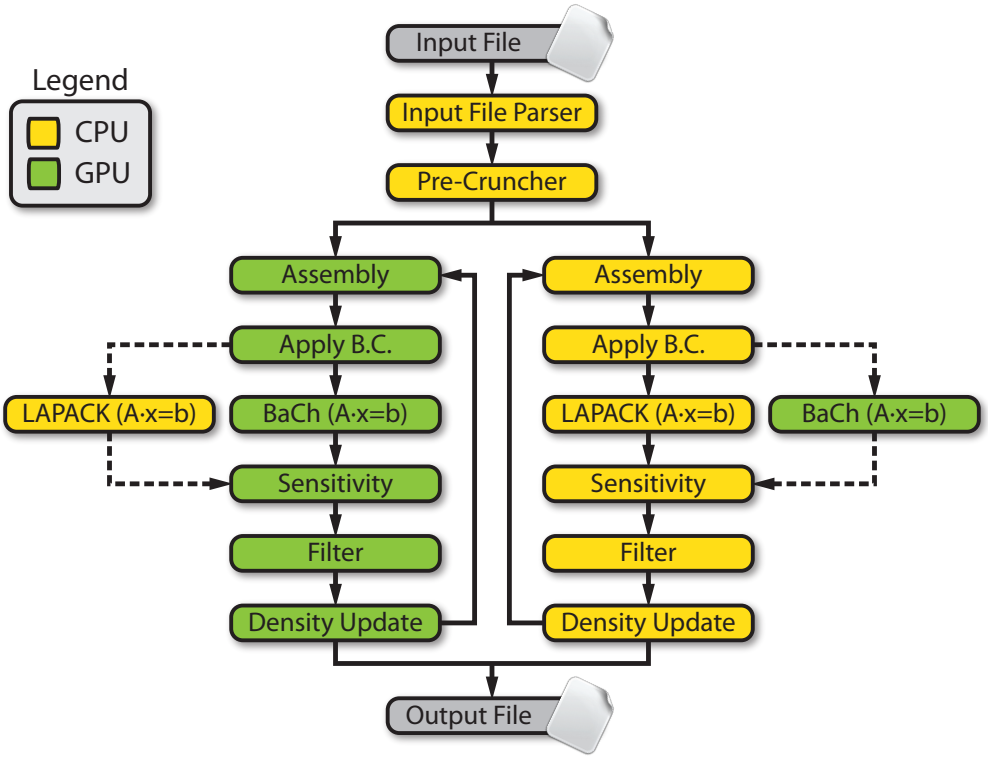


Figure 1.3: Schematic flowchart of the topology optimization code developed for this work.

Work has been done using GPUs to speed the FEM routine [27, 22, 17, 11, 8]. Concurrently, there have been large efforts into GPU based linear solvers [45, 7, 50, 76]. Topology optimization, being a computational intensive task has already been parallelized on traditional arquitectures [28, 49] and on GPUs for structured meshes [39]. The GPU architecture is better suited for structured problems, or linear system formulation where the matrix follows some structure (banded, tri-banded, block-banded and others). The aim of this work is to explore the feasibility and develop a topology optimization code for unstructured 2D meshes on GPUs.

1.1 Thesis Organization

This thesis is organized as follows: the background information of the finite element method and topology optimization is discussed in Chapter 2. The development and testing of a direct solver for the GPU is described in Chapter 3. Next, the implementation of a massively parallel FEM assembly code is explained and detailed in Chapter 4, and the same discussion for the sensitivity filter is in Chapter 5. The remaining algorithms required to complete the code are discussed again in a massively parallel approach in Chapter 6. Examples and benchmarks are run, compared and analyzed in Chapter 7, as well as some code profiling. Finally, in Chapter 8, a summary and conclusion are given with suggestions for the extension of this work in the future.

Chapter 2

Finite Element Method and Topology Optimization Review

The Finite Element Method (abbreviated FEM, or FEA for Finite Element Analysis) is a numerical technique used to solve partial differential equations (PDE) [9, 23, 69, 47, 38, 13]. Because of the nature of numerical method some error in the solution exists (aside from other errors such as model errors due to assumptions, discretization, relaxation and others [58]). One of the primary advantages of FEM over other methods such as the Finite Difference Method (FDM) is the ability to solve the PDE (Partial Differential Equation) over complicated domains: curved boundaries, different problem scales and sizes and even moving boundaries.

The FEM development began around 1940, but it was not until the development of the digital computers that the technique showed its true potential. Soon after, the method quickly acquired popularity due to its flexibility and power, and it currently is one of the most powerful methods in the scientists toolbox [55, 70, 60]. Moreover, many popular software codes make use of it [73, 74, 75, 81, 83].

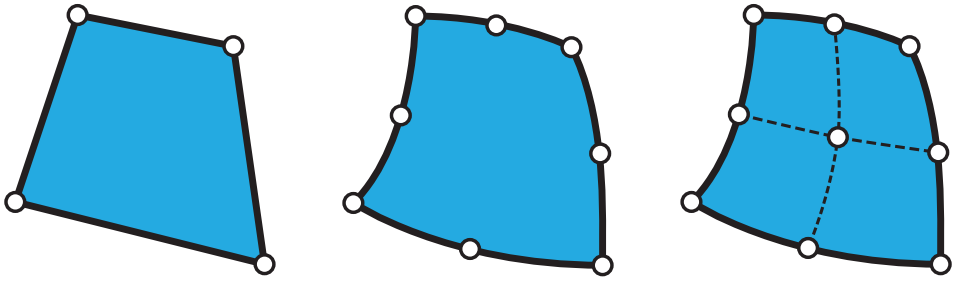
Topology optimization (abbreviated TOP), is a mathematical approach that, making use of FEM, optimizes the material layout (or some other variable) within a design space for a specific problem [54, 68, 42]. The technique attempts to obtain the best structure [30, 59] given a specific objective function (or performance quantification). There was no commercial software that employed this technique until very recently (compared to the many years FEM software has been around) [72, 77, 84, 85, 86].

This chapter aims to be an introduction or review for people who are not familiar with these methods, as they will be extensively used throughout this document.

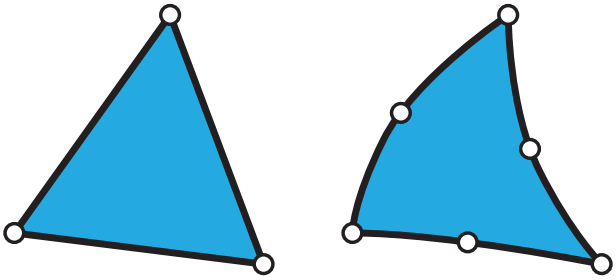
2.1 Finite Element Formulation for Two-Dimensional Solids

Numerical methods in general require the domain to be discretized in some way or another. The FEM typically subdivides the domain in triangular or quadrangular elements for $\Re^2$, and 3D representations of these for domains in $\Re^3$ (cubes, prisms, tetrahedra and others) [23]. The finite elements that are typically

used for 2D problems are illustrated in Figure (2.1).



(a) 4 node quadrangle or Q4 (b) 8 node quadrangle or Q8 (c) 9 node quadrangle or Q9

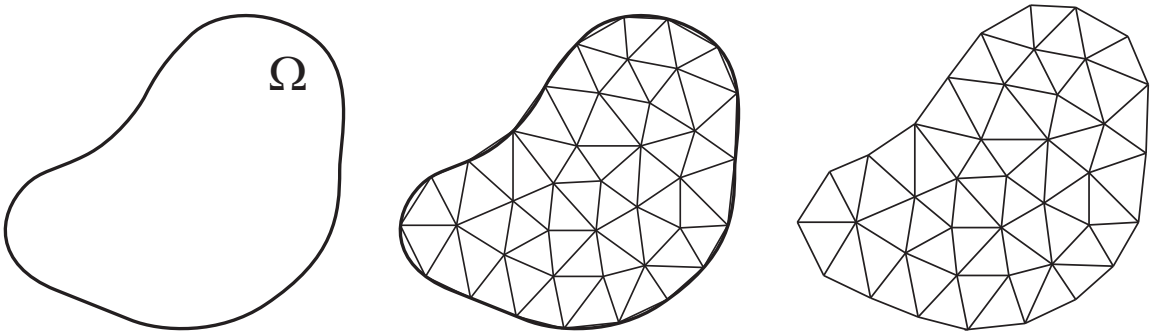


(d) 3 node triangle or T3 (also called CST) (e) 6 node triangle or T6

Figure 2.1: Typical finite elements used in a 2D FEM discretization.

All of these elements have strengths and weaknesses. The strength of the Q4 and T3 (Figures (2.1(a)) and (2.1(d))) is the simplicity of the formulation and meshing, but have poor solution quality, specially for the derivative of the solution field. The T3 is often called *constant strain triangle* or *constant stress triangle*, abreviated CST, because the derivative (of displacement) will be constant throughout the element. Higher order elements like the Q8, Q9 and T6 (Figures (2.1(b)), (2.1(c)) and (2.1(e))) are able to curve their edges, and give a better discretization for domains with curved boundaries, and are also able to represent a solution of higher quality. This comes with a cost due to the more complicated discretization and formulation. Elements of higher order can be constructed, but the ones presented here are the most typical ones.

The domain, generally represented by Ω, will be subdivided into many smaller finite elements: this process is called *meshing*. An example of this can be seen in Figure (2.2). The domain (Figure (2.2(a))) will be subdivided into several finite elements (Figure (2.2(b))), and it is this finite element mesh what will actually be solved by the method (Figure (2.2(c))).



(a) Sample domain Ω (b) Discretization of Ω (c) Mesh representation of Ω

Figure 2.2: Domain discretization using T3 elements.

The FEM formulation for 2D solids is generally written and solved for the displacement variable u in the domain Ω. Because FEM is a numerical technique, we will approximate this continuous u by a piecewise defined field $\hat{u}$, interpolating values from a discrete number of points in the domain (or nodes). All displacements in the domain will be grouped together as a column vector $\tilde{u}$. The displacement inside each element will be a linear combination of these $\tilde{u}$ values at the nodes, done by the so called *shape functions* or N. The solution for the entire domain will be the summation of the solutions for each element. The displacement at any point within element e will be given by:

$$u \approx \hat{u} = \sum_a N_a \tilde{u}_a^e = \begin{bmatrix} N_1 & N_2 & \dots \end{bmatrix} \begin{Bmatrix} \tilde{u}_1 \\ \tilde{u}_2 \\ \vdots \end{Bmatrix}^e = N\tilde{u}^e \tag{2.1}$$

There is freedom to choose the shape functions, but must not violate the following requirement, also called *partition of unity* (some special cases of FEM might not comply with this):

$$\sum_a N_a(x,y) = 1 \qquad \forall (x,y) \in \Omega \tag{2.2}$$

In solids, strains can be computed at any point given the displacements. The relationship can be written as:

$$\varepsilon = \mathfrak{L}u \tag{2.3}$$

where $\mathfrak{L}$ is a suitable differential operator. Because the values in the vector $\tilde{u}$ are constant values over the domain, the differential operator only has effect over the shape functions N. The approximate strain field $\hat{\varepsilon}$ obtained from FEM is:

$$\varepsilon \approx \hat{\varepsilon} = (\mathfrak{L}N)\,\tilde{u} = B\tilde{u} \tag{2.4}$$

The matrix $B = \mathfrak{L}N$ relates displacements with strains. The differential operator $\mathfrak{L}$ is obtained from the relationship between strain and displacements:

$$\varepsilon = \begin{Bmatrix} \varepsilon_x \\ \varepsilon_y \\ \gamma_{xy} \end{Bmatrix} = \begin{Bmatrix} \frac{\partial u}{\partial x} \\ \frac{\partial v}{\partial y} \\ \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \end{Bmatrix} = \begin{bmatrix} \frac{\partial}{\partial x} & 0 \\ 0 & \frac{\partial}{\partial y} \\ \frac{\partial}{\partial y} & \frac{\partial}{\partial x} \end{bmatrix} \begin{Bmatrix} u \\ v \end{Bmatrix} \tag{2.5}$$

In a linear formulation for solids, the relationship between stresses and strains will be of the form:

$$\sigma = D\,(\varepsilon - \varepsilon_0) + \sigma_0 \tag{2.6}$$

where D is an elasticity matrix with the appropriate material properties [55], ε_0 denotes initial strains and σ_0 denotes initial stress. If we keep the same notation used for strains (Equation (2.5)), the stress vector is:

$$\sigma = \begin{Bmatrix} \sigma_x \\ \sigma_y \\ \tau_{xy} \end{Bmatrix} \tag{2.7}$$

The D matrix for a linear isotropic material can be obtained from the typical stress-strain relationship. For the specific case of plane stress it is given by:

$$D = \frac{E}{1-\nu^2} \begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1-\nu}{2} \end{bmatrix} \tag{2.8}$$

and for plane strain it is:

$$D = \frac{E}{(1+\nu)\,(1-2\nu)} \begin{bmatrix} 1-\nu & \nu & 0 \\ \nu & 1-\nu & 0 \\ 0 & 0 & \frac{1-2\nu}{2} \end{bmatrix} \tag{2.9}$$

where E denotes the material's Young modulus, and ν is the Poisson ratio.

The FEM formulation for solid mechanics can be derived in a couple of different ways. Here we present

one of the simplest and most typical one [70]. Taking an arbitrary virtual displacement $\delta\tilde{u}^e$ for the nodes of an element, one obtains the expressions for displacement and strain throughout the element:

$$\delta u = N\delta\tilde{u}^e \tag{2.10}$$

$$\delta\varepsilon = B\delta\tilde{u}^e \tag{2.11}$$

The virtual external work that comes from the nodal forces q_i^e (acting on the direction of displacement i of element e) and the virtual displacements is:

$$\delta W_{ext} = \delta\tilde{u}_1^{eT} q_1^e + \delta\tilde{u}_2^{eT} q_2^e + \cdots = \delta\tilde{u}^{eT} q^e \tag{2.12}$$

Similarly, for this virtual displacement, the internal work per unit volume has two contributions; The stresses σ, and the distributed body forces b:

$$\delta W_{int} = \delta\varepsilon^T\sigma - \delta u^T b \tag{2.13}$$

Using Equation (2.11), one can write the internal work in terms of displacements only:

$$\delta W_{int} = \delta\tilde{u}^{eT}\left(B^T\sigma - N^T b\right) \tag{2.14}$$

The external work has to be equal to the internal work over the entire volume of the element Ω_e:

$$\delta\tilde{u}^{eT} q^e = \delta\tilde{u}^{eT}\left(\int_{\Omega_e} B^T\sigma d\Omega - \int_{\Omega_e} N^T b d\Omega\right) \tag{2.15}$$

This has to valid for any given virtual displacement (excluding the trivial solution when $\delta\tilde{u}^e = 0$). With Equations (2.6) and (2.4), one can express the formulation in terms of our discrete approximation of the displacement field $\tilde{u}^e$,

$$q^e = \int_{\Omega_e} B^T DB d\Omega\tilde{u}^e - \int_{\Omega_e} B^T D\varepsilon_0 d\Omega + \int_{\Omega_e} B^T\sigma_0 d\Omega - \int_{\Omega_e} N^T b d\Omega \tag{2.16}$$

Equation (2.16) can be rewritten as:

$$q^e = k^e\tilde{u}^e + f^e \tag{2.17}$$

where

$$k^e = \int_{\Omega_e} B^T D B d\Omega \tag{2.18}$$

and

$$f^e = -\int_{\Omega_e} N^T b d\Omega - \int_{\Omega_e} B^T D \varepsilon_0 d\Omega + \int_{\Omega_e} B^T \sigma_0 d\Omega \tag{2.19}$$

Loading in the boundary of the domain requires some special treatment. If the boundary is subjected to a distributed external loading $\bar{t}$ per unit area (traction) on the boundary face Γ_e, this boundary load will be represented by equivalent loads on the nodes of the element. Again making use of virtual work, one redefines the external work expression to not only include nodal loads, but distributed on the boundary as well:

$$f^e \to f^e - \int_{\Gamma_e} N^T \bar{t} d\Gamma \tag{2.20}$$

The final force vector f^e results to be:

$$f^e = -\int_{\Omega_e} N^T b d\Omega - \int_{\Omega_e} B^T D \varepsilon_0 d\Omega + \int_{\Omega_e} B^T \sigma_0 d\Omega - \int_{\Gamma_e} N^T \bar{t} d\Gamma \tag{2.21}$$

It is interesting to understand the physical meaning for each component in the force vector f^e. The force vector as written in Equation (2.21) receives contributions from body force, initial strain, initial stress and boundary loads, in that order.

The FEM formulation derived so far by virtual work is applicable to $\Re^n$. For the case of a 2D formulation, the expressions (specially the integrals) can be further simplified. This will use of one of the two D matrices from Equations (2.8) or (2.9), depending on the specific problem to be solved. Additionally, if one assumes that the physical quantities don't change with the thickness of the element t, we can rewrite all the volume integrals, to only integrate the area of the element:

$$\int_{\Omega_e} (\cdot)\, d\Omega = \int_{A_e} (\cdot)\, t dA \tag{2.22}$$

Note that the thickness t can vary over the element.

If every element in the mesh contributes to the global equilibrium, one can *assemble* them all in one equilibrium equation for all the nodal displacements in the mesh:

$$K\tilde{u} + F = q \tag{2.23}$$

where

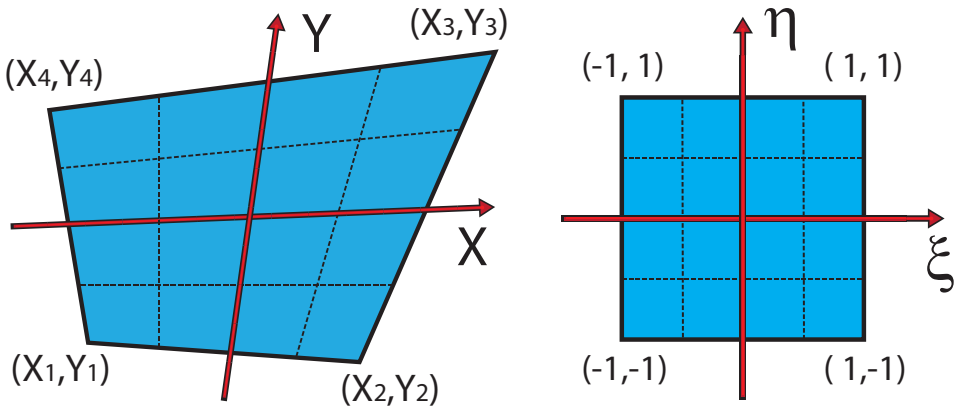
$$K_{ab} = \sum_e k_{ab}^e \tag{2.24}$$

$$F_a = \sum_e f_a^e \tag{2.25}$$

The process of integrating all elements in the equilibrium equation for the entire domain Ω is called *assembly*. Equation (2.23) is solvable for $\tilde{u}$, provided that we can assemble the global stiffness matrix and force vectors. A property of the stiffness matrix K is that for linear elasticity, it will always be symmetric and positive definite. Matrix K will be semi-positive definite if the system has a rigid body motion (improper boundary conditions), or if the structural system has a mechanism of any kind (movable part).

2.2 Isoparametric Formulation

The formulation presented before is valid for any element, in any dimension. But there is a major problem at integrating Equations (2.18) and (2.21) because for each element, the integration domain Ω_e will be different. To solve this, we would like to map these elements to a regular shape that has always the same size and dimensions [12, 21], and in order to maintain the same result for the integral, we have to include a transformation Jacobian.



(a) Cartesian coordinates (b) Local coordinates

Figure 2.3: Two dimensional mapping of a Q4 element.

Local coordinates ξ, η and ζ are called *parent coordinates* (for $\Re^2$ we only use ξ and η). These parent coordinates will be distorted to become x, y and z through some mapping function, and result into what is

called the *mapped element*. For convenience and simplicity, it is desired that our parent element to always span between $-1 \leq \xi, \eta, \zeta \leq 1$. The mapping relations will be of the form:

$$\begin{Bmatrix} x \\ y \\ z \end{Bmatrix} = \begin{Bmatrix} f_x(\xi, \eta, \zeta) \\ f_y(\xi, \eta, \zeta) \\ f_z(\xi, \eta, \zeta) \end{Bmatrix} \tag{2.26}$$

For the case of a 2D element, specifically a Q4 element, if one decides to define the elements by its nodes, with the numbering in a counter-clockwise fashion as illustrated in Figure (2.3(a)), then we expect those nodes to match nodes in the parent element (also in a counter-clockwise fashion) as illustrated in Figure (2.3(b)). Rotation of the mapping is allowed, but to keep the same sign convention for the Jacobian, both parent and mapped element must be numbered in a counter-clockwise fashion.

A convenient way of doing the mapping is to use some kind of shape functions $N'(\xi, \eta)$, but defined in the local coordinate system. Remember that these shape functions have a value of unity for the specific node they belong to, and zero for the other nodes. We can then write the mapping for each element as:

$$\begin{aligned} x &= N'_1 x_1 + N'_2 x_2 + \cdots = N' \begin{Bmatrix} x_1 \\ x_2 \\ \vdots \end{Bmatrix} = N'x \\ y &= N'_1 y_1 + N'_2 y_2 + \cdots = N' \begin{Bmatrix} y_1 \\ y_2 \\ \vdots \end{Bmatrix} = N'y \\ z &= N'_1 z_1 + N'_2 z_2 + \cdots = N' \begin{Bmatrix} z_1 \\ z_2 \\ \vdots \end{Bmatrix} = N'z \end{aligned} \tag{2.27}$$

For the case of a Q4 element, as defined in Figure mapping, these shape functions are:

$$\begin{aligned} N'_1(\xi, \eta) &= \frac{1}{4}(\xi - 1)(\eta - 1) \\ N'_2(\xi, \eta) &= -\frac{1}{4}(\xi + 1)(\eta - 1) \\ N'_3(\xi, \eta) &= \frac{1}{4}(\xi + 1)(\eta + 1) \\ N'_4(\xi, \eta) &= -\frac{1}{4}(\xi - 1)(\eta + 1) \end{aligned} \tag{2.28}$$

The mapping from parent element to mapped element can be easily verified. Taking the x coordinate for node 2 as an example:

$$\left[N'_{\xi=1,\eta=-1}\right]\{x\} = \begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{Bmatrix} = x_2 \tag{2.29}$$

In FEM there is no restriction in the selection of shape functions for the displacement field u, and the shape functions that define geometry. Nevertheless, typically we will use the same shape functions for both fields, and this is the main idea behind the isoparametric formulation:

$$N = N' \tag{2.30}$$

The FEM formulation makes extensive use of the shape function derivatives, and now we need new expressions for these in the parent element coordinate system since we are trying to formulate everything in terms of ξ, η and ζ. By the usual rules of partial differentiation and chain rule, the derivative of a shape function in terms of the parent element coordinates ξ is:

$$\frac{\partial N_a}{\partial \xi} = \frac{\partial N_a}{\partial x}\frac{\partial x}{\partial \xi} + \frac{\partial N_a}{\partial y}\frac{\partial y}{\partial \xi} + \frac{\partial N_a}{\partial z}\frac{\partial z}{\partial \xi} \tag{2.31}$$

Repeating the same procedure for the other two parent coordinates, and reordering in matrix form:

$$\begin{Bmatrix} \frac{\partial N_a}{\partial \xi} \\ \frac{\partial N_a}{\partial \eta} \\ \frac{\partial N_a}{\partial \zeta} \end{Bmatrix} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} & \frac{\partial z}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} & \frac{\partial z}{\partial \eta} \\ \frac{\partial x}{\partial \zeta} & \frac{\partial y}{\partial \zeta} & \frac{\partial z}{\partial \zeta} \end{bmatrix} \begin{Bmatrix} \frac{\partial N_a}{\partial x} \\ \frac{\partial N_a}{\partial y} \\ \frac{\partial N_a}{\partial z} \end{Bmatrix} = [J] \begin{Bmatrix} \frac{\partial N_a}{\partial x} \\ \frac{\partial N_a}{\partial y} \\ \frac{\partial N_a}{\partial z} \end{Bmatrix} \tag{2.32}$$

Matrix $[J]$ is known as the *Jacobian matrix* for the transformation. Specifically, it is desired to write the global derivatives, in function of the local coordinates ξ, η and ζ. Inverting J one obtains:

$$\begin{Bmatrix} \frac{\partial N_a}{\partial x} \\ \frac{\partial N_a}{\partial y} \\ \frac{\partial N_a}{\partial z} \end{Bmatrix} = [J]^{-1} \begin{Bmatrix} \frac{\partial N_a}{\partial \xi} \\ \frac{\partial N_a}{\partial \eta} \\ \frac{\partial N_a}{\partial \zeta} \end{Bmatrix} \tag{2.33}$$

Note that for the transformation to be valid, the *implicit function theorem* requires the inverse of $[J]$ to exist. This is the same as $|J| \neq 0$. The coordinate transformation, as it was defined in Equation (2.27) is

$x(\xi,\eta,\zeta) = \sum_a N'_a(\xi,\eta,\zeta)\, x_a$. The Jacobian matrix is then:

$$[J] = \begin{bmatrix} \sum_a \frac{\partial N'_a}{\partial \xi} x_a & \sum_a \frac{\partial N'_a}{\partial \xi} y_a & \sum_a \frac{\partial N'_a}{\partial \xi} z_a \\ \sum_a \frac{\partial N'_a}{\partial \eta} x_a & \sum_a \frac{\partial N'_a}{\partial \eta} y_a & \sum_a \frac{\partial N'_a}{\partial \eta} z_a \\ \sum_a \frac{\partial N'_a}{\partial \zeta} x_a & \sum_a \frac{\partial N'_a}{\partial \zeta} y_a & \sum_a \frac{\partial N'_a}{\partial \zeta} z_a \end{bmatrix}$$

$$[J] = \begin{bmatrix} \frac{\partial N'_1}{\partial \xi} & \frac{\partial N'_2}{\partial \xi} & \cdots \\ \frac{\partial N'_1}{\partial \eta} & \frac{\partial N'_2}{\partial \eta} & \cdots \\ \frac{\partial N'_1}{\partial \zeta} & \frac{\partial N'_2}{\partial \zeta} & \cdots \end{bmatrix} \begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ \vdots & \vdots & \vdots \end{bmatrix} \tag{2.34}$$

For the 2D case, we drop all the rows and columns containing z and/or ζ from Equation (2.34).

We can now rewrite the integrals for any function $G(x,y,z)$ as:

$$\int_{\Omega_e} G(x,y,z)\, d\Omega_e = \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} \bar{G}(\xi,\eta,\zeta)\, |J(\xi,\eta,\zeta)|\, d\xi\, d\eta\, d\zeta \tag{2.35}$$

where

$$G(x,y,z) = G(x(\xi,\eta,\zeta), y(\xi,\eta,\zeta), z(\xi,\eta,\zeta)) = \bar{G}(\xi,\eta,\zeta)$$

2.3 Numerical Integration - Gauss Quadrature

We have managed to map all integrals to $-1 \leq \xi,\eta,\zeta \leq 1$. Nevertheless, to algebraically integrate this expressions is very expensive. The cost of algebraical integration would render FEM unpractical for any decently sized problem with anything more than just a few elements. Algebraic software processors (better known as Computer Algebra Systems or CAS) such as Mathematica, MuPAD, Maple and others, require several thousands of computer cycles to integrate relatively simple expressions. Thus, we require to evaluate the integrals in some cheap, efficient and hopefully precise manner. This can be achieved exploiting the fact that the integration limits are now the same for all elements, and that the expressions to be integrate are known and polynomial in nature.

The idea of a numerical integration is to evaluate the polynomial expression at certain sampling points, and from linear combination of those values, compute the numerical result of the integral. Several numerical integration rules or techniques exist, but the most common and powerful one is the *Gauss quadrature* or Gauss integration [58, 18]. Gauss quadrature specifically aims for best accuracy at a given number of sampling points.

It is easier to derive the Gauss quadrature rules in $\Re^1$, and then extend to $\Re^n$. Sampling points located at

ξ_i, will be weighted by w_i, and all of them will be added to obtain the desired result:

$$I = \int_{-1}^{1} f(\xi)\, d\xi = \sum_{i=1}^{n} f(\xi_i)\, w_i \tag{2.36}$$

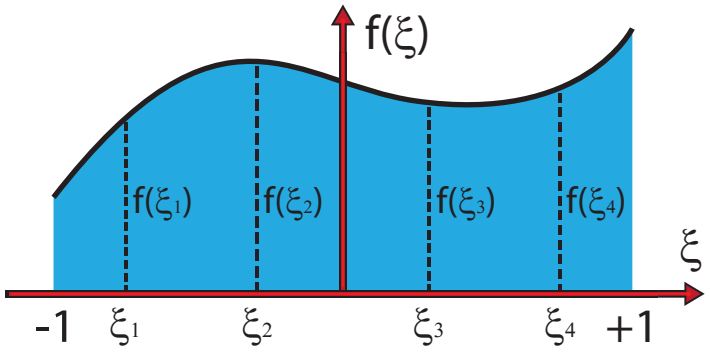


Figure 2.4: Function evaluation locations for the 4-point gauss quadrature.

Each sampling point has a position ξ_i and a weight w_i (example of the 4-point rule in Figure (2.4)). That results in $2n$ unknowns for n sampling points. A polynomial of degree $2n-1$ has exactly $2n$ coefficients. Thus a n-point integration rule, is able to exactly integrate a polynomial of order $2n-1$. The derivation for the 2 point rule will be done as an example. That is:

$$I = \int_{-1}^{1} f(\xi)\, d\xi = f(\xi_1)\, w_1 + f(\xi_2)\, w_2 \tag{2.37}$$

This rule is able to integrate exactly a 3^{rd} order polynomial:

$$\begin{aligned} I &= \int_{-1}^{1} \left(a\xi^3 + b\xi^2 + c\xi + d\right) d\xi = \left[\frac{a}{4}\xi^4 + \frac{b}{3}\xi^3 + \frac{c}{2}\xi^2 + d\xi\right]_{-1}^{+1} \\ I &= \frac{2b}{3} + 2d \end{aligned} \tag{2.38}$$

Note that all terms corresponding to odd powers (after integrating) cancel thanks to the integration limits -1 and 1. A Gauss quadrature characteristic is that it can exactly integrate higher polynomials when compared to other numerical integration rules (other common quadrature rules are Newton-Cotes, Gauss-Konrod and Clenshaw-Curtis). The disadvantage of the Gauss quadrature is that each rule is completely different from the previous one: they are not *nested* (not a progression), where the points and weights from a lower order rule are kept (Gauss-Konrod rules nest). The advantage of nested rules is that if more precision is required, one can add new points and increase the integral accuracy without discarding previous

computations. In Gauss quadrature instead, we have to re-evaluate the entire integral with a higher order rule (new point locations and weights). Other quadrature rules fix the locations of the points and solve for the weights only, making them very simple to derive compared to Gauss quadrature, but making them a lot less precise for the same number of points.

In our 2-point rule derivation, the locations of the gauss points (sampling points), can be obtained by exploiting the symmetry of the integral: where $\xi_1 = \xi_2$ and $w_1 = w_2$:

$$\begin{aligned} I &= f(\xi_1) w_1 + f(\xi_2) w_2 = w \{f(-\xi) + f(\xi)\} \\ I &= w \left\{-a\xi^3 + b\xi^2 - c\xi + d + a\xi^3 + b\xi^2 + c\xi + d\right\} \\ I &= w \left\{2b\xi^2 + 2d\right\} \end{aligned} \tag{2.39}$$

From the results of Equations (2.38) and (2.39), one obtains:

$$\begin{aligned} \frac{2b}{3} + 2d &= w \left\{2b\xi^2 + 2d\right\} \\ 2bw\xi^2 = \frac{2b}{3} \qquad & \qquad 2dw = 2d \end{aligned} \tag{2.40}$$

The weights and locations for the 2-point gauss quadrature rule are:

$$\begin{aligned} \xi_1 &= -\frac{1}{\sqrt{3}} \qquad & w_1 &= 1 \\ \xi_2 &= +\frac{1}{\sqrt{3}} \qquad & w_2 &= 1 \end{aligned} \tag{2.41}$$

Repeating the process for different number of points, we can obtain the weights and locations for any number of points. Table (2.1) has the locations and weights for the first 5 rules, with the 1, 2 and 3 point rules being the most typical ones. Rules of 4, 5 or higher points are only required for high order elements. It should be noted that the one point rule is the trapezoidal integration (it can exactly integrate order $2 \cdot n - 1 = 1$, or a line).

For the case of $\Re^2$ or $\Re^3$, we first want to note that the integrals are commutative:

$$I = \int_{-1}^{1} \int_{-1}^{1} f(\xi, \eta)\, d\xi d\eta = \int_{-1}^{1} \int_{-1}^{1} f(\xi, \eta)\, d\eta d\xi \tag{2.42}$$

To integrate this expression, one first integrates over one variable, and then over the other. That is the same as evaluating one integral keeping the other variable constant. Integrating ξ keeping η constant one obtains:

Table 2.1: Gaussian quadrature weights and locations for the first 5 rules.

Rule Order	Locations ξ_i	Weights w_i
1	$\xi_1 = 0$	$w_1 = 2$
2	$\xi_1 = -1/\sqrt{3}$	$w_1 = 1$
	$\xi_2 = +1/\sqrt{3}$	$w_2 = 1$
3	$\xi_1 = -\sqrt{3/5}$	$w_1 = 5/9$
	$\xi_2 = 0$	$w_2 = 8/9$
	$\xi_3 = +\sqrt{3/5}$	$w_3 = 5/9$
4	$\xi_1 = -\sqrt{\left(3+2\sqrt{6/5}\right)/7}$	$w_1 = \frac{18-\sqrt{30}}{36}$
	$\xi_2 = -\sqrt{\left(3-2\sqrt{6/5}\right)/7}$	$w_2 = \frac{18+\sqrt{30}}{36}$
	$\xi_3 = +\sqrt{\left(3-2\sqrt{6/5}\right)/7}$	$w_3 = \frac{18+\sqrt{30}}{36}$
	$\xi_4 = +\sqrt{\left(3+2\sqrt{6/5}\right)/7}$	$w_4 = \frac{18-\sqrt{30}}{36}$
5	$\xi_1 = -\frac{1}{3}\sqrt{5+2\sqrt{10/7}}$	$w_1 = \frac{322-13\sqrt{70}}{900}$
	$\xi_2 = -\frac{1}{3}\sqrt{5-2\sqrt{10/7}}$	$w_2 = \frac{322+13\sqrt{70}}{900}$
	$\xi_3 = 0$	$w_3 = \frac{128}{225}$
	$\xi_4 = +\frac{1}{3}\sqrt{5-2\sqrt{10/7}}$	$w_4 = \frac{322+13\sqrt{70}}{900}$
	$\xi_5 = +\frac{1}{3}\sqrt{5+2\sqrt{10/7}}$	$w_5 = \frac{322-13\sqrt{70}}{900}$

$$\int_{-1}^{1} f(\xi,\eta)\, d\xi = \sum_{i=1}^{n} f(\xi_i,\eta)\, w_i = g(\eta) \tag{2.43}$$

Integrating the resulting function of η to account for the second integral:

$$I = \int_{-1}^{1}\int_{-1}^{1} f(\xi,\eta)\, d\xi d\eta = \int_{-1}^{1} g(\eta)\, d\eta = \sum_{j=1}^{n} g(\eta_j)\, w_j$$
$$I = \sum_{j=1}^{n}\sum_{i=1}^{n} f(\xi_i,\eta_j)\, w_i w_j \tag{2.44}$$

The integral by the summation of the Gauss points and weights preserves the commutativity of the integrals as expected, and it also says that the same result can be obtained if we integrate η keeping ξ constant. For the case of $\Re^3$:

$$I = \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} f(\xi,\eta,\zeta)\, d\xi d\eta d\zeta = \sum_{k=1}^{n}\sum_{j=1}^{n}\sum_{i=1}^{n} f(\xi_i,\eta_j,\zeta_k)\, w_i w_j w_k \tag{2.45}$$

For $\Re^m$, the n point integration rules, becomes the n^m rule. For example, in $\Re^2$, the 3-point rule is called the 3×3 rule. For the case of a Q4 element, it can be exactly integrated using the 2×2 rule.

It is possible to use a lower-than-required integration rule for an element. While not exact, the result will approximate the exact value. This is sometimes done to save computation time, or for special purposes like reducing the effect of *shear locking*. Nevertheless, this will also introduce a problem in the formulation that might manifest called *spurious zero energy modes*, or SZEM for short [60, 53].

The mechanism of SZEM can be explained when we try to integrate a 2^{nd} order polynomial with a 1-point rule. Since we evaluate the function at 1 point only, there is an infinite number of 2^{nd} order polynomials that contain that single point (located at $\xi = 0$), and all of these have completely different results when integrated from -1 to 1. This translates in an inability of the 1-point rule to detect these changes (the Gauss quadrature result is the same even when the function is able to change). In solid mechanics this means that the element can undergo deformation requiring zero energy (hence the name). These deformation modes don't always manifest, and usually require to be triggered by some loading condition (dynamic problems often suffer from this).

2.4 Putting It All Together

We finally have a mechanical procedure (therefore ideal for computing), to obtain the stiffness matrix and force vectors for any element, but in our case we will focus and detail them for the Q4 element.

The displacement field $\hat{u}$ in $\Re^2$ has two components, that we will call u and v, that is displacement in the x and y directions respectively, where u_1 and v_1 correspond to the horizontal and vertical displacement for node 1. Using the shape functions to interpolate within the element, one obtainst:

$$\hat{u} = \begin{Bmatrix} u \\ v \end{Bmatrix} = \begin{bmatrix} N_1 & 0 & N_2 & 0 & N_3 & 0 & N_4 & 0 \\ 0 & N_1 & 0 & N_2 & 0 & N_3 & 0 & N_4 \end{bmatrix} \begin{Bmatrix} u_1 \\ v_1 \\ u_2 \\ v_2 \\ u_3 \\ v_3 \\ u_4 \\ v_4 \end{Bmatrix} \tag{2.46}$$

With the displacement defined, we can now apply the differential operator $\mathfrak{L}$ to get $B = \mathfrak{L}N$. With that, we get the strains:

$$\hat{\varepsilon} = \begin{Bmatrix} \varepsilon_x \\ \varepsilon_y \\ \gamma_{xy} \end{Bmatrix} = [B]\,\hat{u}$$

or, equivalently

$$\hat{\varepsilon} = \begin{bmatrix} \frac{\partial}{\partial x} & 0 \\ 0 & \frac{\partial}{\partial y} \\ \frac{\partial}{\partial y} & \frac{\partial}{\partial x} \end{bmatrix} \begin{bmatrix} N_1 & 0 & N_2 & 0 & N_3 & 0 & N_4 & 0 \\ 0 & N_1 & 0 & N_2 & 0 & N_3 & 0 & N_4 \end{bmatrix} \begin{Bmatrix} u_1 \\ v_1 \\ u_2 \\ v_2 \\ u_3 \\ v_3 \\ u_4 \\ v_4 \end{Bmatrix}$$

The strains in cartesian formulation are:

$$\hat{\varepsilon} = \begin{bmatrix} \frac{\partial N_1}{\partial x} & 0 & \frac{\partial N_2}{\partial x} & 0 & \frac{\partial N_3}{\partial x} & 0 & \frac{\partial N_4}{\partial x} & 0 \\ 0 & \frac{\partial N_1}{\partial y} & 0 & \frac{\partial N_2}{\partial y} & 0 & \frac{\partial N_3}{\partial y} & 0 & \frac{\partial N_4}{\partial y} \\ \frac{\partial N_1}{\partial y} & \frac{\partial N_1}{\partial x} & \frac{\partial N_2}{\partial y} & \frac{\partial N_2}{\partial x} & \frac{\partial N_3}{\partial y} & \frac{\partial N_3}{\partial x} & \frac{\partial N_4}{\partial y} & \frac{\partial N_4}{\partial x} \end{bmatrix} \begin{Bmatrix} u_1 \\ v_1 \\ u_2 \\ v_2 \\ u_3 \\ v_3 \\ u_4 \\ v_4 \end{Bmatrix} \tag{2.47}$$

, and in the scope of an isoparametric formulation:

$$\begin{bmatrix} \frac{\partial N_a}{\partial x_b} \end{bmatrix} = [J]^{-1} \begin{bmatrix} \frac{\partial N_a}{\partial \xi_b} \end{bmatrix}$$

$$\begin{bmatrix} \frac{\partial N_1}{\partial x} & \frac{\partial N_2}{\partial x} & \frac{\partial N_3}{\partial x} & \frac{\partial N_4}{\partial x} \\ \frac{\partial N_1}{\partial y} & \frac{\partial N_2}{\partial y} & \frac{\partial N_3}{\partial y} & \frac{\partial N_4}{\partial y} \end{bmatrix} = [J]^{-1} \begin{bmatrix} \frac{\partial N_1}{\partial \xi} & \frac{\partial N_2}{\partial \xi} & \frac{\partial N_3}{\partial \xi} & \frac{\partial N_4}{\partial \xi} \\ \frac{\partial N_1}{\partial \eta} & \frac{\partial N_2}{\partial \eta} & \frac{\partial N_3}{\partial \eta} & \frac{\partial N_4}{\partial \eta} \end{bmatrix} \tag{2.48}$$

Note that Equation (2.47) is simply a rearrangement of the derivatives obtained from Equation (2.48). In an isoparametric formulation, typically $N' = N$, and then the transformation Jacobian for element e is (with nodes numbered in counter-clockwise fashion):

$$[J^e] = \begin{bmatrix} \frac{\partial N_1}{\partial \xi} & \frac{\partial N_2}{\partial \xi} & \frac{\partial N_3}{\partial \xi} & \frac{\partial N_4}{\partial \xi} \\ \frac{\partial N_1}{\partial \eta} & \frac{\partial N_2}{\partial \eta} & \frac{\partial N_3}{\partial \eta} & \frac{\partial N_4}{\partial \eta} \end{bmatrix} \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \\ x_4 & y_4 \end{bmatrix} \tag{2.49}$$

The local stiffness matrix for a Q4 element using an isoparametric formulation is:

$$[k^e] = \int_{-1}^{1} \int_{-1}^{1} [B^e]^T [D] [B^e] |J^e| \, d\xi d\eta \tag{2.50}$$

where $[B^e]$ is:

$$[B^e] = [J^e]^{-1} \begin{bmatrix} \frac{\partial N_1}{\partial \xi} & \frac{\partial N_2}{\partial \xi} & \frac{\partial N_3}{\partial \xi} & \frac{\partial N_4}{\partial \xi} \\ \frac{\partial N_1}{\partial \eta} & \frac{\partial N_2}{\partial \eta} & \frac{\partial N_3}{\partial \eta} & \frac{\partial N_4}{\partial \eta} \end{bmatrix} \tag{2.51}$$

Using Gaussian quadrature, and a 2×2 rule that is capable of exactly integrating a Q4 element, the final expression for the stiffness matrix is:

$$[k^e] = \sum_{i=1}^{2} \sum_{j=1}^{2} [B^e(\xi_i, \eta_j)]^T [D] [B^e(\xi_i, \eta_j)] |J(\xi_i, \eta_j)| \, w_i w_j \tag{2.52}$$

The same procedure can be repeated for the boundary loads, body forces and others. Equation (2.52) is the final expression that was used in the present work (and in most of the available FEM software). It should be noted that the resulting local stiffness matrix for the Q4 element will be a symmetric 8×8, positive definite matrix (for any value of $E > 0$, and with a finite element area), if rigid body modes are constrained.

Table 2.2: Shape functions and its derivatives for a Q4 element

Node Number	Shape Function	$\partial N_a / \partial \xi$	$\partial N_a / \partial \eta$
1	$N_1 = \frac{1}{4}(\xi - 1)(\eta - 1)$	$\frac{1}{4}(\eta - 1)$	$\frac{1}{4}(\xi - 1)$
2	$N_2 = -\frac{1}{4}(\xi + 1)(\eta - 1)$	$-\frac{1}{4}(\eta - 1)$	$-\frac{1}{4}(\xi + 1)$
3	$N_3 = \frac{1}{4}(\xi + 1)(\eta + 1)$	$\frac{1}{4}(\eta + 1)$	$\frac{1}{4}(\xi + 1)$
4	$N_4 = -\frac{1}{4}(\xi - 1)(\eta + 1)$	$-\frac{1}{4}(\eta + 1)$	$-\frac{1}{4}(\xi - 1)$

2.5 Topology Optimization

Topology optimization is a mathematical approach that optimizes a material layout for a specific set of loads, boundary conditions and design variable constraints [54, 68, 4, 3]. A topology optimization problem will have a *design space*, *constraints* and *objective function*.

- The design space represents the volume within which the design will take place. When defining this space, we must take into account requirements of accessibility, constructability, and other factors that might limit or restrict our possible design space.
- The constraints are criteria that the design must not violate. The most typical constraint is the volume fraction constraint. This enforces the fact that we want to optimally distribute a finite amount of material (hence prevents the algorithm from making everything solid).
- The objective function is a mathematical expression that will quantify the optimality, (or score) of our current design. This function tells how optimal our structure is compared to another. It is important to note that the algorithm optimizes this specific function, and thus, if the objective function is different, the optimal design will also change to seek the optimum of this new objective function.

For the case of material distribution, we can see the design variable as the material density. We typically want zero-one designs, that is, no intermediate densities (viewed in relative densities, that is 0 for void and 1 for solid). If a variable can only take a 0 or 1 value, the problem then becomes an *integer programming* problem, more specifically a *0-1 integer programming* or *binary integer programming* (BIP). This type of problems have extreme non-linearities, and actually bifurcate with each variable state. There are methods to solve this kind of problems, like the *cutting-plane method*, *branch and bound*, *branch and cut*, and *branch and price* among others. Nevertheless these methods generally cannot handle more than just a few integer variables, and in topology optimization, the number of design variables can be well over a million. The reason for this is that integer programming problems are in fact *NP-hard*[1].

To solve and obtain a solution of the zero-one type, we have to include some relaxation of the problem. That is, replace the original problem formulation with an almost equivalent one that is easier to solve. In our case, this will consist of allowing the design variable to be continuous, and then somehow taylor it towards a zero-one design. The most typical approach to topology optimization is the so-called *power-law approach* or SIMP (Solid Isotropic Material with Penalization) [5, 6]. In this approach, material properties do not change with the relative density, but they are modeled instead as the properties of a solid material times

[1] An NP-hard problem has a difficulty that does not scale in a polynomial manner with size (they usually scale in a factorial or exponential manner). A slightly easier type of problems are *NP-Complete* problems, that while still non polynomial, the solution can be usually verified in an easier way (polynomial time).

the relative density to some power. For the element stiffness for example, one obtains:

$$[k_e] = \rho^p \cdot [k_{e0}] \tag{2.53}$$

where ρ is the relative density, k_0 is the solid material stiffness, and p is called the penalization factor. The penalization factor, as the name implies, penalizes intermediate densities and makes it uneconomical or inefficient for the algorithm to have intermediate densities, hence favoring a zero-one design. The SIMP approach has been criticized because there is almost no material that behaves according to the power-law interpolation (in some special cases, with a specific set of penalization and material properties the material is physically feasible). One should note that if $p \to \infty$ the problem then becomes a binary integer programming problem.

Depending on what is being optimized, the objective function can take several different expressions. Nevertheless, the most common one for material distribution is compliance, that is the product of displacements and forces (internal energy):

$$\begin{aligned} c(\rho) &= \{u\}^T \{f\} = \{u\}^T [K] \{u\} \\ c(\rho) &= \sum_{e=1}^{n} (\rho_e)^p \{u_e\}^T [k_{e0}] \{u_e\} \end{aligned} \tag{2.54}$$

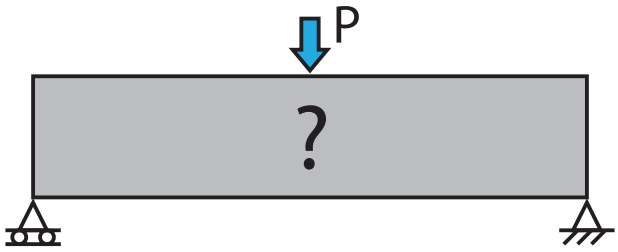
The formulation for the optimization problem that looks for the optimal material distribution, given a limited amount of material, is then:

$$\begin{aligned} \min_{\rho} : \quad & c(\rho) = \sum_{e=1}^{n} (\rho_e)^p \{u_e\}^T [k_{e0}] \{u_e\} \\ \text{subject to} : \quad & [K] \{u\} - \{F\} = 0 \\ & \sum_{e=1}^{n} (\rho_e \cdot V_e) - f \cdot V_0 = 0 \\ & 0 < \rho_{min} \leq \rho \leq 1 \end{aligned} \tag{2.55}$$

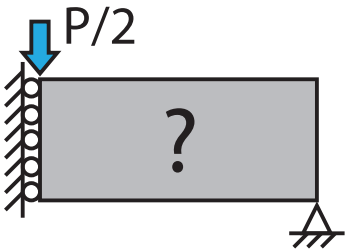
where ρ is the vector of design variables (relative densities), ρ_{min} is a vector of minimum relative densities to keep the problem positive definite and avoid singularities, V_e corresponds to the volume of element e, and f is the specified volume fraction of the total volume $V_0 = \sum V_e$. Typically, the penalization power ranges between 2 and 5, with $p = 3$ being a typical value.

A common application of Topology Optimization is the so called MBB beam (Messerschmitt-Bölkow-Blohm) design problem. In this case we want to design a simply supported beam (Figure (2.5(a))), that due to

symmetry can be simplified to the design of half of the beam (Figure (2.5(b))), and be mirrored to the other half (Figure (2.5(c))).



(a) Messerschmitt-Bölkow-Blohm (MBB) beam design problem.



(b) Symmetric half of the MBB beam.

(c) Design for the MBB beam using Topology Optimization

Figure 2.5: MBB beam problem designed using Topology Optimization (square-element mesh of 320×70, volume fraction $f = 0.5 = 50\%$, penalization $p = 3$ and $r_{min} = 7$).

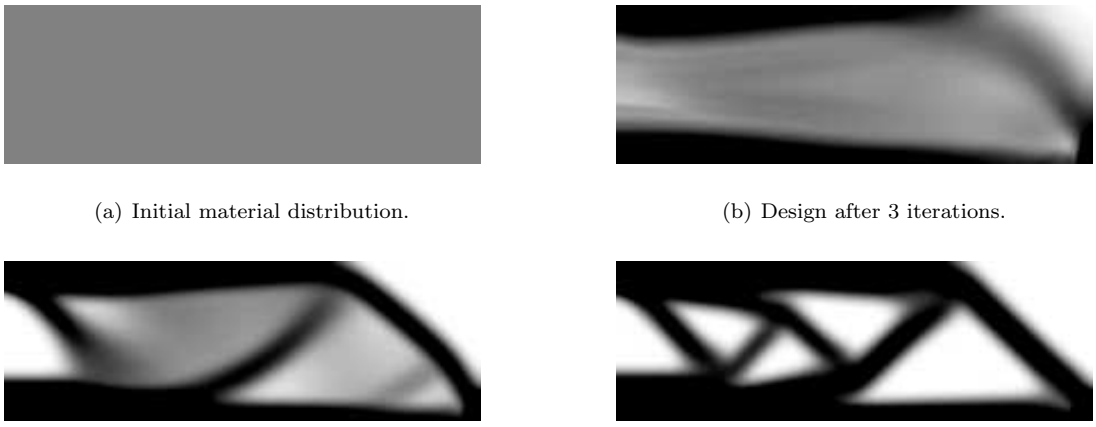
The optimization problem in Equation (2.55) can be solved using different methods, such as Optimality Criteria (OC) [1], Sequential Linear Programming (SLP), Method of Moving Asymptotes (MMA) [44] and others. The advantage of OC is simplicity, but it is most effective when only one constraint is present, in our case, the volume constraint (the optimization problem presented in Equation (2.55) seems to have more than one constraint, but we can further simplify it to only one).

The OC method, like most of them, solves the problem in an iterative manner. The starting point, or initial design is normally assumed to be uniform and equal to the volume fraction f. The topology optimization problem may have multiple local minima, and a different starting design might result in a different

final solution. The topology optimization problem is ill-posed and non-linear, and can have multiple local minima. Because of this, we need to further impose a constraint on the OC method to prevent it from diverging as it tries to make too big strides towards finding the optimum:

$$\rho_e^{new} = \begin{cases} \max\left(\rho_{min}, \rho_e - m\right) & \text{if} \quad \rho_e \cdot B_e^{\eta} \leq \max\left(\rho_{min}, \rho_e - m\right) \\ \rho_e \cdot B_e^{\eta} & \text{if} \quad \max\left(\rho_{min}, \rho_e - m\right) < \rho_e \cdot B_e^{\eta} < \min\left(1, \rho_e + m\right) \\ \min\left(1, \rho_e + m\right) & \text{if} \quad \min\left(1, \rho_e + m\right) \leq \rho_e \cdot B_e^{\eta} \end{cases} \tag{2.56}$$

where m is a positive move-limit, η is a numerical damping to add stability to the method (typically $\eta = 1/2$), and B_e is found from the optimality condition. The progress of these iterations at some steps for half of the MBB beam are shown in Figure (2.6). Also note that, after a sufficient number of iterations (Figure (2.6(d))), the design is closely resembling a zero-one design (no gray elements).



(a) Initial material distribution.

(b) Design after 3 iterations.

(c) Design after 15 iterations.

(d) Design after 50 iterations.

Figure 2.6: Half MBB beam Topology Optimization iterations example.

The optimization problem in its original form is hard to solve. The most widely used technique to find the optima of such inequality-constrained problems is the Lagrange multiplier method [66]. Using Lagrange multipliers, the topology optimization formulation becomes:

$$\begin{aligned} \Lambda\left(\rho, \lambda\right) = & \sum_{e=1}^{n} \left(\rho_e\right)^p \left\{u_e\right\}^T \left[k_{e0}\right] \left\{u_e\right\} + \lambda_1 \left(\sum_{e=1}^{n} \left(\rho_e \cdot V_e\right) - f \cdot V_0\right) \\ & + \left\{\lambda_2\right\}^T \left(\left[K\right] \left\{u\right\} - \left\{f\right\}\right) \end{aligned} \tag{2.57}$$

where $[K] = \sum [k_e]$, with the sum operation in this case indicating assembly. Note that $\{\lambda_2\}^T$ is actually a column vector. The sensitivity of the objective function, can be interpreted as the gain from adding additional material in a specific location. The higher the sensitivity, the more convenient it is to make that design variable solid. Using chain rule, one obtains the sensitivity as:

$$\frac{\partial \Lambda}{\partial \rho_e} = \frac{\partial \Lambda}{\partial \{u_e\}} \frac{\partial \{u_e\}}{\partial \rho_e} + \frac{\partial \Lambda}{\partial [k_e]} \frac{\partial [k_e]}{\partial \rho_e} + \lambda_1 \cdot V_e \tag{2.58}$$

That results in:

$$\begin{aligned} \frac{\partial \Lambda}{\partial \rho_e} = & \left(2 \{u_e\}^T [k_e] + \{\lambda_2\}^T [k_e]\right) \frac{\partial \{u_e\}}{\partial \rho_e} \\ & + \{u_e\}^T \frac{\partial [k_e]}{\partial \rho_e} \{u_e\} + \{\lambda_2\}^T \frac{\partial [k_e]}{\partial \rho_e} \{u_e\} + \lambda_1 \cdot V_e \end{aligned} \tag{2.59}$$

The sensitivity of the stiffness matrix with respect to the design variables $\partial [k_e] / \partial \rho_e$ can be directly obtained from the SIMP model formulation since

$$[k_e] = [k_e (E (\rho_e))]$$

but for the displacement, that is $\partial \{u_e\} / \partial \rho_e$, it is not straightforward. Two techniques are mainly used to solve this problem: the *direct method* and the *adjoint method* [57]. They both have advantages and disadvantages, and the selection of one or the other will depend on the specific problem. For the case of compliance with volume constraint the ideal technique is the adjoint method. Solving the adjoint problem, we conclude that:

$$\{\lambda_2\}^T = -2 \{u_e\}^T \tag{2.60}$$

The dependency of the sensitivity with $\partial \{u_e\} / \partial \rho_e$ gets canceled, and the expression finally comes to:

$$\frac{\partial \Lambda}{\partial \rho_e} = - \{u_e\}^T \frac{\partial [k_e]}{\partial \rho_e} \{u_e\} + \lambda_1 \cdot V_e \tag{2.61}$$

and at the optimum:

$$\frac{\partial \Lambda}{\partial \rho_e} = 0 \tag{2.62}$$

This agrees with the definition of an optimum, where we cannot improve the quality of our solution in one area without worsening some other. Viewed in the frame of topology optimization, it is the state where

the material is at its best possible location. Equation (2.62) can be rearranged, resulting in:

$$1 = \frac{\{u_e\}^T \frac{\partial [k_e]}{\partial \rho_e} \{u_e\}}{\lambda_1 \cdot V_e} \tag{2.63}$$

For the case when we are not at the optimum, this expression will be different from 1, and that will be called B_e. Note that if $B_e \gg 1$, it is then very convenient to add material in element e. The relationship between the stiffness and the relative density is given by Equation (2.53). The expression then becomes:

$$B_e = \frac{p \cdot \rho_e^{(p-1)} \{u_e\}^T [k_{e0}] \{u_e\}}{\lambda_1 \cdot V_e} = \frac{\frac{\partial c}{\partial \rho_e}}{\lambda_1 \cdot V_e} \tag{2.64}$$

with,

$$\frac{\partial c}{\partial \rho_e} = p (\rho_e)^{p-1} \{u_e\}^T [k_e] \{u_e\} \tag{2.65}$$

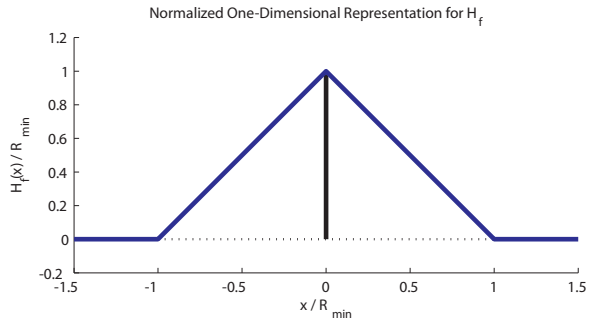
The Lagrangian multiplier λ must be such that the new densities obtained from Equation (2.56) conform to the volume constraint $\sum (\rho_e \cdot V_e) - f \cdot V_0 = 0$. An easy way to obtain λ is by a *binary search algorithm.* Topology optimization formulations, like the one presented here, typically suffer from two types of problems: mesh dependency and very stiff artificial structures like checkerboards. While checkerboarding can be solved using higher order elements or a different design variable approach such as CAMD (Continuous Approximation of Material Distribution) [29] or MTOP (Multiresolution Topology Optimization) [32], both problems are solved using a filtering technique [54, 42]. In simple words, the filter blurs the sensitivities of the mesh (same as a convolution). For instance, the filter modifies the element sensitivities:

$$\frac{\hat{\partial} c}{\partial \rho_e} = \left(\frac{1}{\rho_e V_e \sum_{f=1}^{n} \hat{H}_f} \right) \sum_{f=1}^{n} \hat{H}_f \rho_f V_f \frac{\partial c}{\partial \rho_f} \tag{2.66}$$

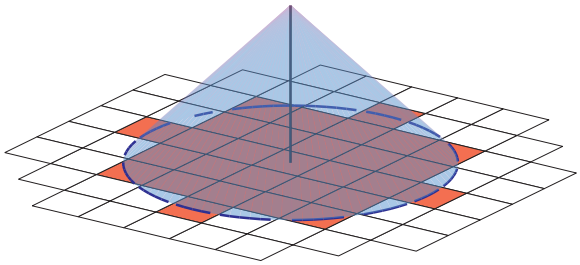
With the convolution operator (weight factor) $\hat{H}_f$ typically defined as:

$$\hat{H}_f = \begin{cases} r_{min} - \text{dist}(e, f) & \text{if} \quad \text{dist}(e, f) \leq r_{min} \\ 0 & \text{if} \quad \text{dist}(e, f) > r_{min} \end{cases} \tag{2.67}$$

where r_min is the filter radius, $\text{dist}(e, f)$ is the distance between the locations of the design variables for elements e and f (typically at the center of the element). In 1D this wighting function looks like a triangle centered on element e decaying linearly (Figure (2.7(a))), and in 2D it looks like a cone (Figure (2.7(b))). In some cases, different convolution operators (non-linear decay) are used to achieve different filter behaviors. These new modified sensitivities defined by Equation (2.66) will be used instead of the original ones to avoid mesh dependency and checkerboarding. As an example, half of the MBB beam problem was run with and



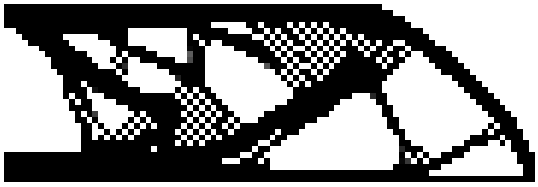
(a) One-dimensional representation of $\hat{H}_f$.



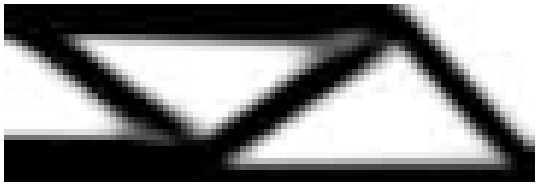
(b) Two-dimensional representation of $\hat{H}_f$.

Figure 2.7: Convolution operator $\hat{H}_f$ for one and two-dimensional cases.

without sensitivities filtering and the results are in Figure (2.8). In these figure the checkerboard pattern is present only in the unfiltered version of the problem as expected.



(a) Half MBB beam with no filtering.



(b) Half MBB beam with filtering.

Figure 2.8: Half MBB beam example, showing the effect of the filter in the final design (square-element mesh of 90×30, volume fraction $f = 0.5 = 50\%$, penalization $p = 3$ and $r_{min} = 3$).

The topology optimization iterations can go on indefinitely unless some stop criteria is used. Since this is a multivariable problem, norms are typically used. The *p-norm* is the most general of all, and can be used to derive other simpler more specific ones:

$$\|x\|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p} \tag{2.68}$$

where $p \geq 1$, with a special case when $p = \infty$:

$$\|x\|_\infty = \max x_i$$

A common stopping criteria in topology optimization is to monitor the max change of relative density from one iteration to another, or in norm language, the *infinity norm* of the change in densities. The algo-

rithm stops once the norm is below some specified tolerance as:

$$\|\rho_e^{new} - \rho_e\|_\infty = \max\left(\rho_e^{new} - \rho_e\right) < \text{tolerance} \tag{2.69}$$

Another type of norm, a bit less common is the *1-norm* (this is the same as taking $p = 1$ in Equation (2.68)), also called *taxicab norm* or *manhattan norm*. This norm has the disadvantage of being mesh-dependent (it grows with the number of elements), but that can easily be solved dividing by the number of elements in the mesh. A stopping criteria using the 1-norm is:

$$\frac{\|\rho_e^{new} - \rho_e\|_1}{n} = \frac{\sum_{e=1}^{n} |\rho_e^{new} - \rho_e|}{n} < \text{tolerance} \tag{2.70}$$

Other stopping conditions may be used or explored, nevertheless the ones presented here are the most common ones.

This completes the topology optimization algorithm: an iterative method that makes use of FEM analysis to obtain optimal structures. One must keep in mind that there are many local minima, and there is no guarantee that the solution found is actually a global minima. Also, the final design will depend on the objective function that was chosen (typically compliance, but others may be used). Moreover, the use of filters may steer the designs to something considered to be more feasible, but generally off from the optimum. The final design is not really a zero-one design, and some interpretation is required, but if a sufficiently high penalization is used (but still small enough to keep the algorithm stable), there should be little left to interpretation.

Chapter 3

BaCh Solver

In computational mechanics, many problems ultimately result in solving a system of linear equations, and sometimes the system has to be solved several times (dynamic, nonlinearities, optimization problems and others). Hence, a small increase of speed in the system solving routine can have a major impact in currently available codes with minimal modification. Some attempts to achieve this are already available to the public but at the time of this writing, they are either still research codes [50, 45, 7] or commercial ones [76]. Additionally, due to the fact that GPU computing is a relatively new field, specialized solvers such as banded, symmetric, complex and others are not common nor readily available, even in commercial packages. In some cases, the matrices are well-structured and the fills follow a simple pattern, and a very efficient solver can be applied [41, 39]. We want to develop a **Ba**nded Symmetric **Ch**olesky-based Positive Definite **Solver** (refered as BaCh Solver from here on), that can be used as a drop in replacement for a CPU equivalent implementation available in LAPACK [78], more specifically in ACML [71].

The BaCh solver aims to be almost a drop-in replacement for LAPACK's **spbsv** solver (details for this solver can be found in the LAPACK manuals [52]). It is not a full replacement because of some limitations in the current code, and some minor differences in the output. These limitations and differences will be discussed later.

3.1 Data Handling and Storage

A banded symmetric system of equations of the form $A \cdot x = b$ will have a matrix A with a structure as shown in Figure (3.1). Because the matrix is symmetric, only half of it is stored. If the bandwidth is relatively small, this will result in a very efficient storage method that is very competitive when compared to some simple sparse matrix storage schemes, and usually faster and easier to read. In practical problems, high level of storage and compute efficiency can be achieved when bandwidth reduction schemes are used

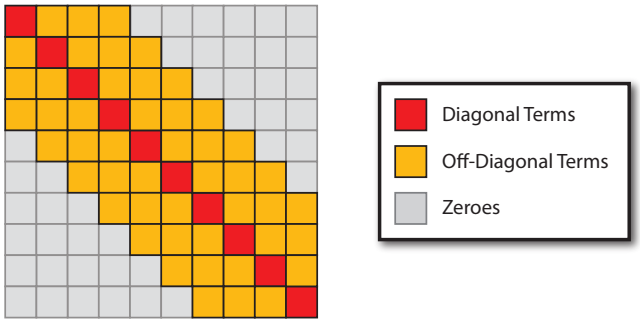


Figure 3.1: Schematical structure for a banded symmetric matrix.

[10, 26, 15, 16, 36, 37].

The storage scheme used by the BaCh solver is the same as in LAPACK when called from C/C++. Because LAPACK was originally written in FORTRAN 77 (now it is written in FORTRAN 90) its preferred array storage method is the uncommon *column-major method* (programming languages that use this scheme are FORTRAN and MATLAB). This storage scheme will actually result in a performance hit for the BaCh solver due to some memory access conflicts in the GPU architecture. FORTRAN uses 1-based indexing, as opposed to C/C++ 0-based indexing. When calling LAPACK from C/C++ the 1-based indexing is not carried from FORTRAN, but the column-major storage method still is.

The banded storage maps a one dimensional square array to store the upper or lower band. Figure (3.2) illustrates how the squared array gets mapped to the original banded matrix if lower triangular storage is used. Note that there is always a triangle of unused information. If a banded symmetric matrix A is stored in

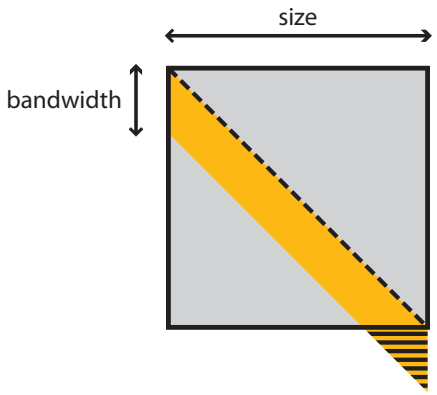


Figure 3.2: Squared array mapped to store a symmetric banded matrix.

a 1D array called *data*, the mapping using column-major storage and 0-based indexing for a lower triangular

storage scheme is:

$$A_{i,j} = A_{j,i} = \text{data}\left[bandwidth \cdot j + (i - j)\right] \tag{3.1}$$

with $i \geq j$, $i - j < bandwidth$ and $j < size$. Figure (3.3) shows the indexing for this compact storage method for a matrix with $size = 7$ and $bandwidth = 4$. The current implementation of the solver can only

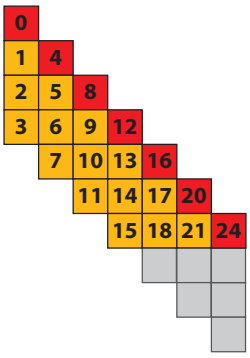


Figure 3.3: Symmetric banded storage indexing for a matrix with $size = 7$ and $bandwidth = 4$.

handle a single column in the left hand side of the linear system, i.e. the column vector b. With all these considerations the storage is relatively simple and straightforward.

3.2 Implementation

A symmetric system can be solved using the Cholesky decomposition [58, 66, 67]. If a matrix A has a Cholesky decomposition (i.e. is positive definite), that means there is a lower triangular matrix L such that:

$$A = L \cdot L^T \tag{3.2}$$

The Cholesky decomposition can now be used to solve the linear system of equations $A \cdot x = b$ first by solving $L \cdot y = b$ and then $L^T \cdot x = y$. Note that L^T is an upper triangular matrix, and the solution to the system can be obtained with two substitution procedures: First by a forward substitution for the lower triangular system, and then a backward substitution for the upper triangular system.

The following two equations are used to compute the Cholesky decomposition of matrix A:

$$L_{i,i} = \sqrt{A_{i,i} - \sum_{k=1}^{i-1} L_{i,k}^2} \tag{3.3}$$

$$L_{i,j} = \frac{A_{i,j} - \sum_{k=1}^{j-1} L_{i,k} \cdot L_{j,k}}{L_{j,j}} \tag{3.4}$$

From these equations it can be derived that if matrix A is banded, then the lower triangular matrix L is also banded and has the same bandwidth A has.

The forward substitution expression is:

$$x_i = \frac{b_i - \sum_{k=1}^{i-1} L_{i,k} \cdot x_k}{L_{i,i}} \tag{3.5}$$

Since the Cholesky decomposition traverses the matrix in a "forward" fashion, just like the forward substitution, it makes sense to do both at the same time while traversing the stored data forward. This way, we can afford to only traverse the data twice in order to solve the system, once for the Cholesky decomposition and forward substitution, and a second time for the backward substitution. Traversing data in the GPU should be minimized as much as possible in order to achieve a high throughput.

The BaCh solver launches $(bandwidth - 1)$ threads. A thread is an independent unit of processing within a computer. In simpler words, a thread executes commands and instructions, and several threads can execute these concurrently since they are independent from each other. Thread number 0 has the special task of handling diagonal entries, and this one along with the rest of the threads, all handle the off diagonal entries [51]. Figure (3.4) illustrates the row each thread is assigned to. Because the current implementation of CUDA (Version 2.3) can launch a maximum of 512 threads, the maximum bandwidth size the BaCh solver in its current implementation is able to handle is $bandwidth = 513$. However, this limitation can be overcome as the software and hardware develop.

A simplified pseudo-code that obtains the Cholesky decomposition of matrix A in a parallel fashion, where

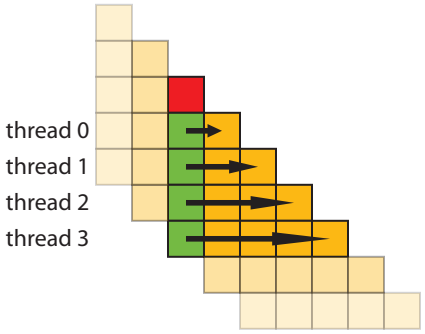


Figure 3.4: Thread assignment for the Cholesky decomposition and forward substitution process.

b_w denotes the bandwidth, n the matrix size and $thrID$ is the thread ID number is presented in Pseudo-code

(1). Some details are missing, and special care should be taken for the last b_w columns, because the number of available rows in those columns is smaller than the bandwidth and constantly decreasing.

Pseudo-code 1 Simplified parallel *Cholesky* and *Forward substitution*

```
if (thrID==0) {
  SharedDiag = sqrt(A[0]);
  A[0] = SharedDiag;
  SharedSol = b[0] / SharedDiag;
  b[0] = SharedSol;
}
for (i=1; i<n; ++i) {
  synchronize();
  MyValue = A[(i-1)*bw+thrID+1] / SharedDiag;
  A[(i-1)*bw+thrID+1] = MyValue;
  SharedCol[threadID] = MyValue;
  b[i+thrID] -= MyValue * SharedSol;

  synchronize();
  if (thrID==0) {
    A[(i-1)*bw] = CurrentDiag;
    CurrentDiag = sqrt(A[i*bw]-MyValue^2);
    SharedSol = b[i] / CurrentDiag;
    b[i] = SharedSol;
  }
  else {
    for (j=0; j<thrID; ++j)
      A[(i+j)*bw+thrID-j] -= MyValue * SharedCol[j];
    A[(i+thrID)*bw] -= MyValue^2;
  }
}
```

After the first part of the code obtains the Cholesky decomposition and does the forward substitution, all that is required is the backward substitution to get the solution of the system. Pseudo-code (2) illustrates the parallel backward substitution: again, special care should be taken for the last b_w columns.

Pseudo-code 2 Simplified parallel *Backward substitution*

```
if (thrID==0) {
  SharedSol = b[n-1] / A[(n-1)*bw];
  b[n-1] = SharedSol;
}
for (i=n-2; i>=0; --i) {
  synchronize();
  b[i-thrID] -= A[(i-thrID)*bw+thrID+1] * SharedSol;

  synchronize();
  if (thrID==0) {
    SharedSol = b[i] / A[i*bw];
    b[i] = SharedSol;
  }
}
```

The backward substitution is much simpler, and the only key difference is that the calls to matrix A (now containing the Cholesky decomposition), are done as if it was an upper triangular matrix (calling a

transposed version of A).

Compared to the simplified code presented here, the actual solver makes more use of shared memory in order to reduce the number of global memory accesses. This due to memory access being one of the most important bottlenecks that should be overcome in order to achieve high throughput in a parallel code [65]. Nevertheless, the amount of data that can be placed in shared memory, and therefore recycled within a column stride is not much, and considering that the storage of matrix A follows a column-major format, reading a column of data will likely have memory bank conflicts and or misaligned access pattern [64]. This situation will limit the speedup we will be able to obtain from the algorithm.

The BaCh solver requires the data to be copied to the device (or GPU) before it can be solved. After the solution is obtained, it must be copied back to the CPU. The connection between the CPU memory and the GPU memory is an extremely fast PCI Express connection (at the time of this writing, PCI Express is the fastest off-the-shelf available interface between a computer processor and its components or expansion cards, with the GPU being one of them). Nevertheless, this is something the CPU solver is not required to do. Recent implementations of CUDA and new chips can access CPU memory in an almost direct way (Zero-copy access, introduced in CUDA v2.2).

The functions in LAPACK usually return a flag indicating if the function had errors. If the data is not correctly supplied, or if the matrix is not positive definite, **spbsv** will report the error through this flag. In the case of the BaCh Solver, an error in the data or the case that the matrix is not positive definite can be checked by querying the last value of the solution. In the case of an error, this value will be *NaN*, that is a special value or symbol to denote *Not A Number* in floating point calculations.

3.3 Benchmarks

The BaCh Solver was benchmarked against a multiprocessor implementation of LAPACK, specifically ACML_MP. The hardware used in the benchmarks consist of a dual-socket dual-core AMD Opteron 2216 processors (4 cores in total), 8GB RAM and an NVIDIA Tesla T10 Processor with 4GB.

The runtime for both solvers for 32 different sized problems (Figure (3.5)) and the complexity of the solver algorithm was used to adjust an equation for the runtime of each. The complexity of a Cholesky decomposition is $O\left(b_w^2 \cdot n\right)$ and the complexity of the forward and backward substitutions [58], as well as the task of copying the data to the device (GPU memory) is $O\left(b_w \cdot n\right)$. Considering that there might be a solver initialization time (constant), we introduce the following fit to evaluate runtime:

$$\text{runtime} = a_0 + a_1 \cdot b_w \cdot n + a_2 \cdot b_w^2 \cdot n \tag{3.6}$$

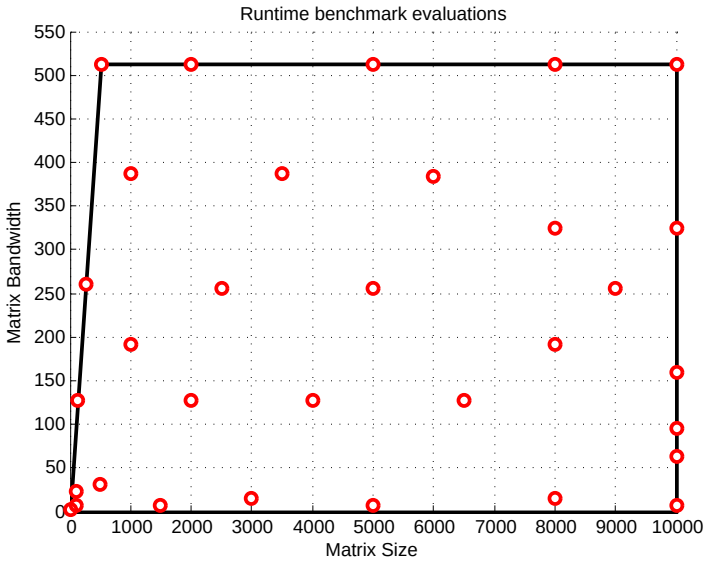


Figure 3.5: Sampling points for the solver runtime approximations.

The benchmarks run in the GPU were done after the CUDA driver has been initialized (there is a driver initialization time with the very first CUDA function call), not to be confused with a kernel launch. Since the vast majority of the applications require to solve systems more than once, the single CUDA driver initialization time can be neglected. The fitted expressions are the following (time in milliseconds):

$$\text{GPU} = 5.96 \cdot 10^{-4} \cdot b_w \cdot n + 5.29 \cdot 10^{-8} \cdot b_w^2 \cdot n \tag{3.7}$$

$$\text{CPU} = 41.07 + 8.76 \cdot 10^{-5} \cdot b_w \cdot n + 1.25 \cdot 10^{-7} \cdot b_w^2 \cdot n \tag{3.8}$$

Thus, the speedup of the GPU solver over the CPU solver can be obtained and plotted (Figure (3.6)). If we plot the contour for which the speedup ratio is equal to one, we can further recognize the area until which the BaCh solver presents an advantage over its LAPACK counterpart (Figure (3.7)).

The solution quality of the solvers is assessed using the relative error. The following expression is used to calculate the relative error ε_{rel}:

$$\varepsilon_{rel} = \frac{x_{solver} - x_{exact}}{x_{exact}} \tag{3.9}$$

In order to analyze the error of the algorithm, 10 different randomly generated positive definite matrices with $n = 10,000$ and $b_w = 256$ were generated with condition numbers ranging from 100 to 2500. Forward and backward relative errors [58] for all 10 problems are plotted on top of each other for the cases of sparse

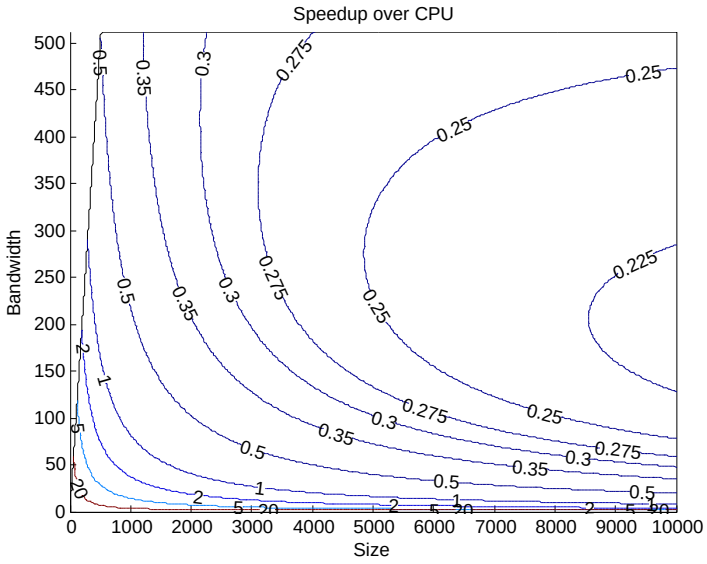


Figure 3.6: Contourplot of the speedup of the GPU over the CPU solver.

MATLAB, LAPACK's *spbsv* and the BaCh solver. Figures (3.8) and (3.9) illustrate the error distribution across the elements of the vectors. The total error is the summation of errors from the forward and backward substitutions. The error from the substitutions grow as the substitution process progresses, and because the forward and backward substitution traverse the elements in opposite directions, the total error has its maximum within the midspan of the elements range. This behavior is exhibited by all solvers tested.

MATLAB is always several magnitudes more precise than the other two solvers mainly due to the fact that it operates in double precision, and will be used as a reference to compare the other solvers. The forward relative error (error in the solution vector) is bigger for the BaCh solver compared to LAPACK. The backward relative error is the error in the left hand side vector using the solution vector that was obtained from the solver. While there is a noticeable difference between both the LAPACK and BaCh solver in the forward error, the backward error is pretty similar for both; relatively small, with a few spikes of higher errors for some elements.

3.4 Remarks

There is a region of decent sized problems for which the BaCh solver outperforms other solvers of the same type available in the literature. Low performance gain from the solver can be attributed to the way data is stored and to the fact that banded systems are employed. GPUs have generally good speedups when dealing with dense matrices [45, 50] or very well behaved diagonal systems [41] and not for sparse. Very

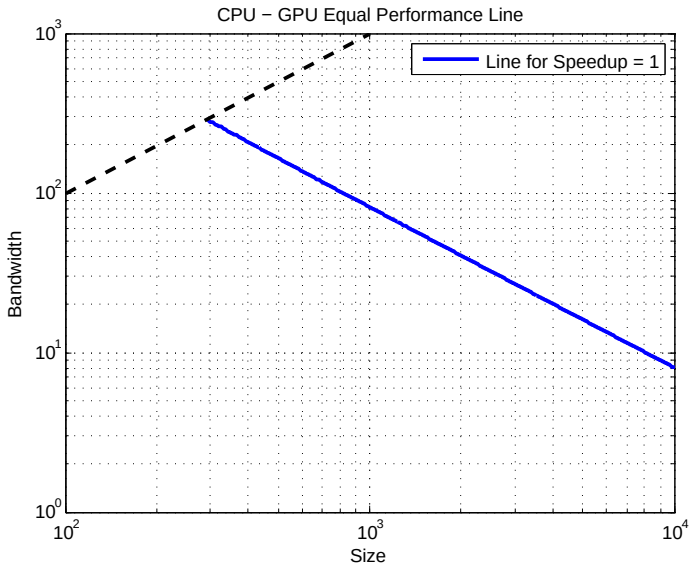


Figure 3.7: Line of GPU/CPU speedup ratio equal to 1.

little progress has been done in sparse algebra for GPUs [2, 48].

Comparing the errors with LAPACK's single precision solver, the difference is small for the relative forward error, and equivalent for the relative backward error. The time that is lost copying the data to the GPU, and back, can be saved if the data is generated within the GPU, thus offering an alternative to CPU solvers that would require the data to be copied from the GPU to the CPU before it can be solved (same problem the BaCh solver has when invoked from the CPU, but in this case it is the other way around).

The BaCh solver presents an alternative to the iterative solvers available for GPUs, and it is efficient compared to a CPU solver for small sized problems, specially for small bandwidths.

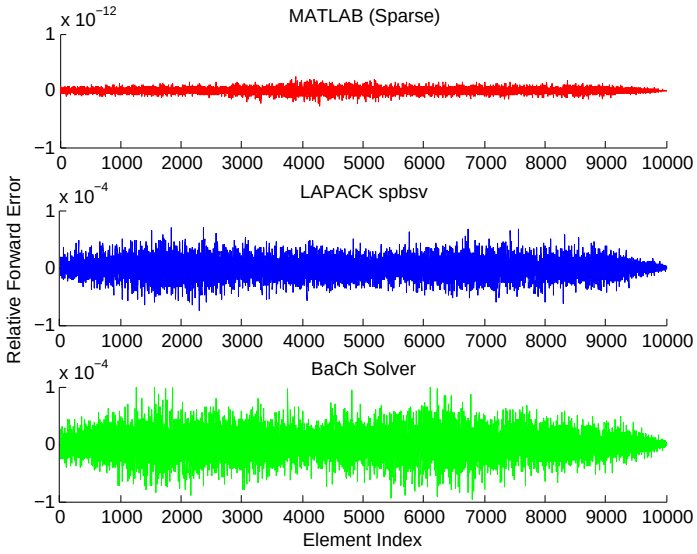


Figure 3.8: Forward relative error for 10 different randomly generated systems.

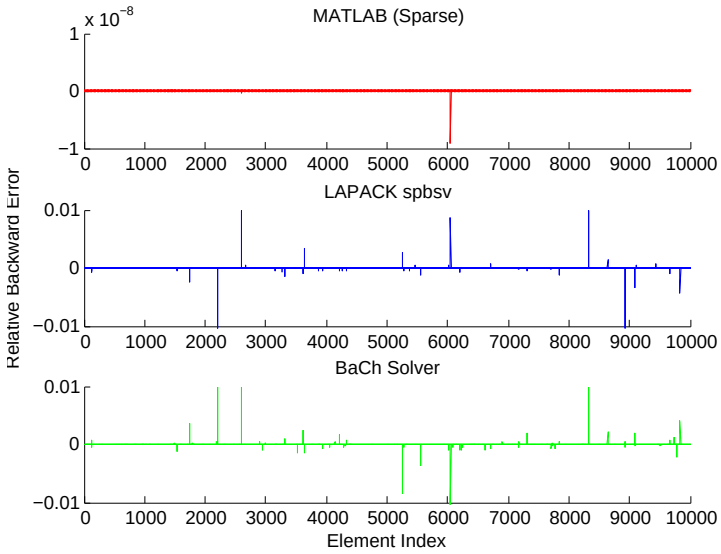


Figure 3.9: Backward relative error for 10 different randomly generated systems.

Chapter 4

Stiffness Assembly and Coloring

Scientific modeling ultimately converges to assembling and solving a system of equations (once or multiple times). Before we can solve the system, some type of assembly has to take place in order to construct the matrix and vector to be solved. In sequential codes, assembling such a system has no specific difficulty, however, in a massively parallel code many problems arise because of events that execute concurrently. The most notorious of these problems is the so called *race condition* (or *race hazard*). A race condition occurs when an algorithm is dependent on the sequence or timing of almost concurrent events. This usually occurs when the device attempts to perform two or more operations at the same time, but in reality the operations must be done in order or sequence to obtain the correct result.

In the presence of a race condition, the algorithm must be either changed or execution must be organized and ordered in such a way that there is no interference between parallel events and the dependence on timing is eliminated. For the specific case of the Finite Element Method applied to solid mechanics, the race condition is generally present in the assembly of the stiffness matrix $[K]$ and the computation of the force vector $\{F\}$, being the assembly of the stiffness matrix the worst case of the two.

There are a few different approaches possible for computing and assembling the local stiffness matrices. Most of these approaches can be divided into two groups: The nodal approach and the element wise approach.

In the nodal approach, each thread executing in parallel computes the local stiffness matrices for all the elements sharing a specific node. Once the stiffness for this node is computed, the value can be safely assembled onto the global stiffness matrix. It is safe to assemble the matrix this way, because each thread is in charge of the positions corresponding to a single node in the stiffness matrix. The problem with this approach is that there is a lot of over computation: each element's local stiffness matrix is computed as many times as it has nodes (for Q4 elements, each element is computed 4 times).

The element approach does not discard any computation, but must be careful enough to make sure that no other thread is assembling to the same nodes at a given time. To fix this, an approach based on graph coloring is studied. Graph coloring is an attractive solution to the problem due to its simplicity and ability

to render decent enough results with little overhead [34, 8].

4.1 Race Condition

Following the element base approach for the assembly of the stiffness matrix, it was derived previously (Equation (2.24)) that the global stiffness matrix $[K]$ is the summation of all the smaller element stiffness matrices $[k]$ in specific locations of $[K]$. This process is called *assembly*, and suffers from severe race condition if special measures are not taken.

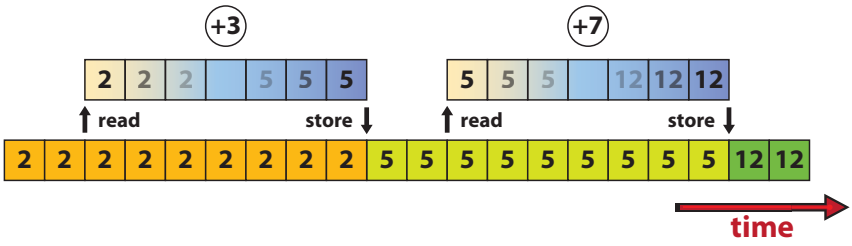
Using index notation, if the degrees of freedom for element i are stored in order in a vector DOF^i, the assembly of this element can be easily written as:

$$K_{DOF^i,DOF^j} = K_{DOF^i,DOF^j} + k_{i,j} \tag{4.1}$$

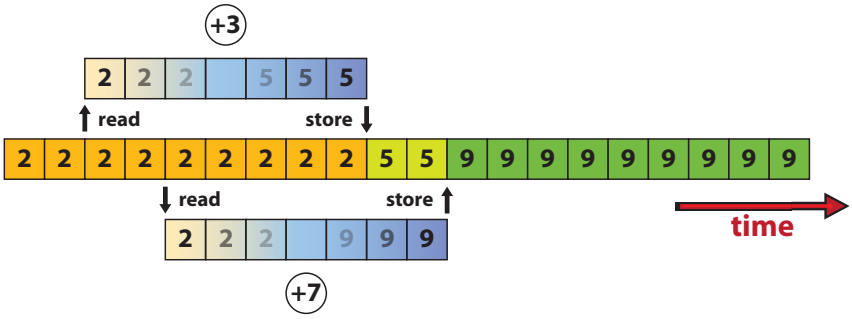
The addition operation is commutative, this means that the order in which the assembly process takes place does not matter (except for rounding errors). Nevertheless, if two additions or assemblies are done concurrently, the result might not be as expected because computers require a certain number of clock cycles (or time steps) to execute any operation. While doing this, the original value stored in memory is copied to a temporal register (or local memory) where the operation will actually take place. Once the operation is over, the register is copied back to memory with the result. Figure (4.1(a)) illustrates how two consecutive additions might be done in a sequential code for $f(x) = x+3+7$ with $x = 2$ in this case (each box is a clock cycle or time step, with addition requiring 5 clock cycles in this example). Because the code is sequential, only one operation takes place at a given time and the result is, as expected, $f(2) = 12$. If one of the additions begins before the last operation finishes, then the result from the first one is lost. Figures (4.1(b)) and (4.1(c)) are examples of how the previous additions might result in an incorrect result for a parallel code prone to race condition. The total execution time for the parallel code is smaller than the sequential code, explaining the possibility of speedup from a parallel code (provided that the race condition is eliminated and the correct result is obtained).

The assembly code for the global stiffness matrix $[K]$ has to make sure that two local stiffness matrices $[k]$ are never assembled (added) to the global stiffness matrix in the same positions at the same time. That is exactly the same as enforcing all index vectors DOF^i to have all different indices at a given step:

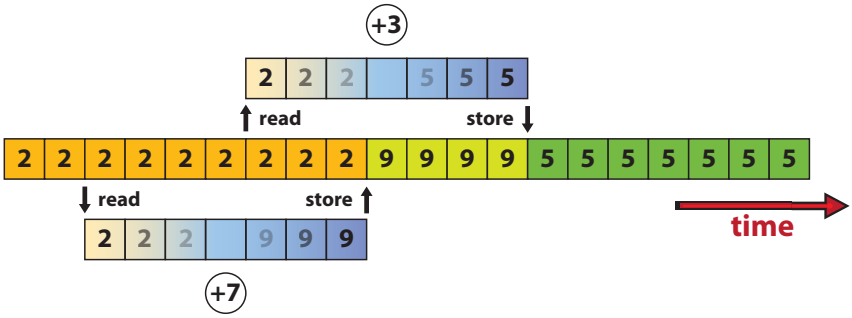
$$\left(DOF^i\right) \cap \left(DOF^j\right) = \emptyset \qquad \forall\, i, j \, with \, i \neq j \tag{4.2}$$



(a) Sequential code obtaining the correct result.



(b) Case 1 of a parallel code obtaining an incorrect result.



(c) Case 2 of a parallel code obtaining an incorrect result.

Figure 4.1: Illustration of simple mathematical operations with and without race condition problems, where each clock cycle or time step is represented by a box. Addition in this case takes 5 clock cycles to compute.

The idea of assembling a specific group of elements that complies with Equation (4.2) in parallel before moving to a new group of elements arises. With enough groups of elements we should be able to assemble the entire global stiffness matrix $[K]$ without race conditions.

4.2 Graph Coloring

If we understand the parallel assembly as a process where no neighboring elements (elements that share one or more nodes) may be assembled within a stride (or step), we can define groups of elements that can be

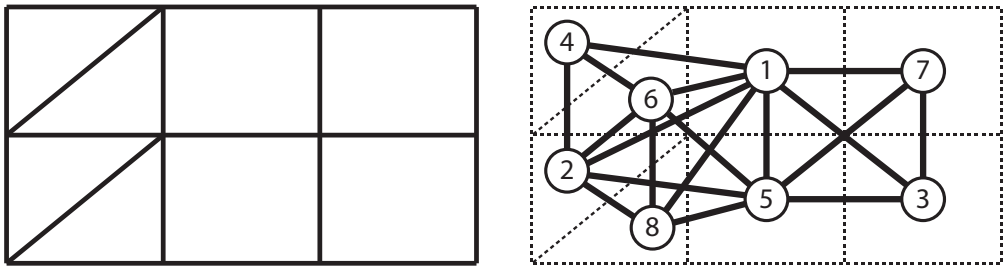
safely assembled at the same step (this groups will be often called *colors*). This is the same concept behind *graph coloring*, or *map coloring*, where the same color cannot be used for neighboring zones [34, 8].

Graph coloring is in fact an NP-Hard problem, that is, the complexity scales in a non-polynomial way. In simple words, making the problem just slightly bigger, makes its solution several times harder to obtain. In most cases, the problem size is such that the optimal solution cannot be found in a reasonable time.

These type of problems (NP-Hard and NP-Complete), are often tackled with heuristic solutions [14]. These are usually good enough solutions that take much less time to obtain. Nothing ensures that these heuristic solutions actually correspond to the optimal solution for the problem, and for most cases will not, but in the vast majority of the problems and considering the difficulty of the problem, these solutions are satisfactory.

The optimal solution for the graph coloring problem is translated in using the fewest number of colors to *paint* the complete graph or map. This optimal number of colors is called the *chromatic number*, and is denoted by $\chi(G)$, for a given graph G. Obtaining a solution that utilizes exactly $\chi(G)$ colors for a typical FEM problem with several thousand elements is unpractical, this because the graph coloring algorithm would take too long to obtain the solution. We need a fast heuristic algorithm that colors the graph in a reasonable time. By reasonable time, we consider it to be relatively small when compared to the computations to be done after this (in our case, the topology optimization loop). The hardware also limits the maximum number of threads, that in graph coloring terms is the maximum number of elements a color can have. Experimentation proved that the proposed heuristic algorithm give closer to optimum solutions when this restriction is active.

The FEM mesh can be translated to a *communication graph*, that stores all the information regarding each element's neighbors. A simple way to represent this graph is through the *communication matrix* for graph G called $L\left(G^C\right)$, where a 1 in position (i, j) of this matrix means that elements i and j are neighbors, and 0 if not. Communication is bidirectional, which means that if element i neighbors j, then j neighbors i as well. The same applies if they are not neighbors ($L_{i,j} = L_{j,i}$), hence the communication matrix is symmetric [36, 37]. Depending on the specific algorithm, the diagonal may be filled with zeroes, ones or used to store the total number of neighbors of each element. If the diagonal is filled with either zeroes or ones, the entire matrix is of *bool* type (zero-one, or binary). For large meshes, the communication matrix is mostly composed of zeroes, and for the case of the algorithm presented here, the diagonal is not used at all. For the case of the present work, a sparse matrix storage is used storing only the positions of the ones, and since the diagonal is not used, it considers the diagonal to be filled with zeroes ($L_{i,i} = 0$), further reducing the storage space. Generating and storing the information contained in $[L]$ requires a dynamically growing data structure and the *Standard Template Library* (STL) [82], included in the C++ Standard Library was used for this task.



(a) FEM Mesh. (b) Elements communication graph.

Figure 4.2: Communication graph for a simple mesh (extracted from [36]).

An example of this scheme can be seen for the mesh in Figure (4.2(a)), where after numbering the elements in any desired order, the equivalent communication graph can be generated as in Figure (4.2(b)). From the communication graph, the communication matrix can be easily derived:

$$L\left(G^C\right) = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

The proposed algorithm that colors this graph is extremely simple and belongs to a family of algorithms known as *Greedy Coloring Algorithms* [14]. The main idea of the algorithm is to color the next element on the list with the first available color that is not in use by a neighbor. Colors can be unavailable for use at a current element if they are either in use by a neighbor or if the color is full (thread limit says that a color cannot have more than $NumberOfThreads$ elements assigned to it). The color information is stored in an integer vector C, where 0 denotes that the element is not colored, and any number $j > 0$ in position i, implies that element i belongs to color j. The thread restriction in the color size can be written using boolean operators:

$$\sum_{i=1}^{e} (C_i == j) \leq NumberOfThreads \qquad \forall j \in Colors \tag{4.3}$$

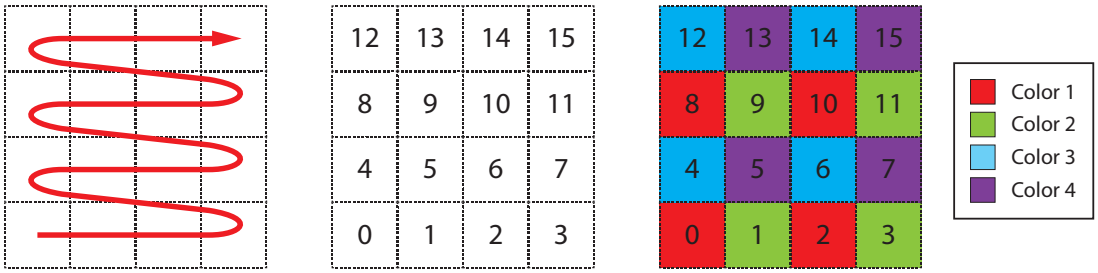
The pseudo-code for the graph coloring algorithm used here is explained in Pseudo-code (3). Some details are omitted for ease of understanding. This code returns a vector C that assigns a color for each element, and $CurrentColor$ has the exact number of colors used for the entire graph.

Pseudo-code 3 Fast and simple *Graph Coloring* algorithm

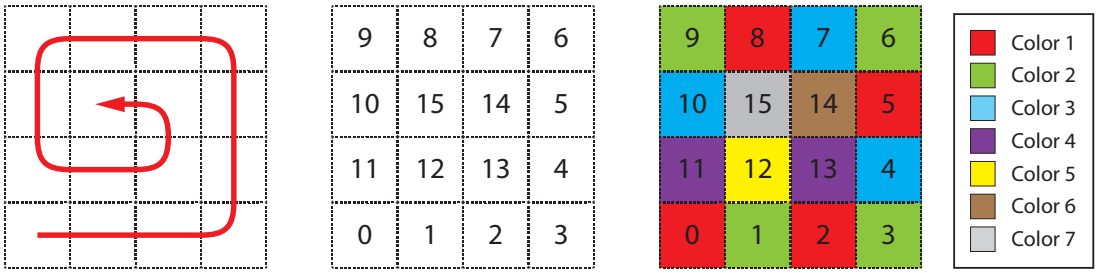
```
fill(ColoredElements,NULL);
fill(FilledColors,false);
fill(IsElementColored,false);
fill(ColorSize,0);

FirstColor=0;
NumberOfColors=1;
for (i=0; i<NumberOfElements; ++i) {
  % Variable i represents the Current Element to be colored
  fill(BlockedColors,false);
  for (j=0; j<Neighbors[i].Number; ++j) {
    ThisNeighbor=Neighbors[i].Neighbor[j];
    if (IsColored[ThisNeighbor]==true)
      BlockedColors[ColoredElements[ThisNeighbor]]=true;
  }
  % Find the first usable color
  for (j=FirstColor; j<NumberOfColors; ++j) {
    if (BlockedColors[j]==false && FilledColors[j]==false)
      break;
  }
  ColoredElements[i]=j;
  IsElementColored[i]=true;
  ++ColorSize[j];
  % Did we just create a new color?
  if (j==NumberOfColors)
    ++NumberOfColors;
  % If old color was used, check if it is now full
  else if (ColorSize[j]==NumberOfThreads) {
    FilledColors[j]=true;
    while (FilledColors[FirstColor]==true)
      ++FirstColor;
  }
}
```

A common characteristic of the greedy family of coloring algorithms is that they are sensitive to the way the data is traversed. This because the algorithm makes no attempt to predict future moves, or *see ahead*. Instead, the majority of the coloring algorithms work in a *first-come first-served* basis. Therefore, the order in which the elements are listed or numbered can have a severe impact on the results [24, 14], specially for small meshes. Using this algorithm, a structured 4×4 mesh is colored, but using three different numbering schemes. Results of this study are available in Figure (4.3), with Figure (4.3(a)) actually resulting in the optimal coloring (using only 4 colors since $\chi(G) = 4$ for this specific case), and Figure (4.3(b)) having a bad outcome with 7 colors. The assembly code (and the coloring algorithm) makes no distinction between

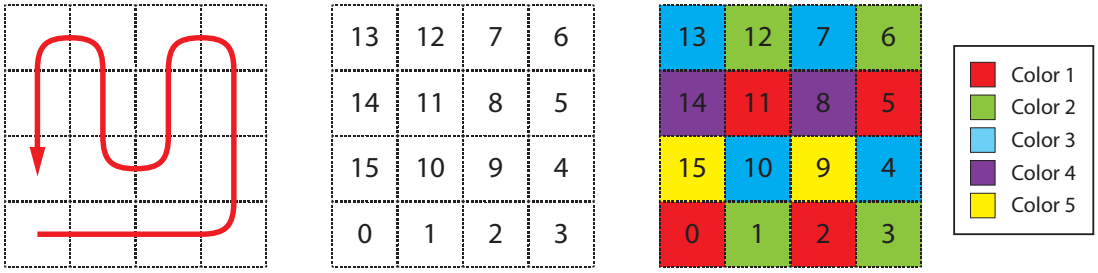
structured and unstructured meshes. The treatment of both structured and unstructured is the same, and both will have an impact on the coloring algorithm's result depending on the numbering sequence.



(a) Numbering scheme resulting in 4 colors.



(b) Numbering scheme resulting in 7 colors.



(c) Numbering scheme resulting in 5 colors.

Figure 4.3: Greedy coloring for three different numbering schemes in a simple 4×4 structured mesh.

A more complex mesh with 708 elements that was colored by the algorithm is presented in Figure (4.4). In this case the maximum number of parallel threads was specified to be 128, and the algorithm solution for this problem uses 12 colors. Each color represents a group of elements that can safely be assembled concurrently in the stiffness matrix for a FEM formulation, without causing a race condition. After each color is assembled, a synchronization barrier has to be placed, in order to be sure that all threads have finished their task before continuing to a new step or stride (in graph coloring terms, move to the next color).

Obtaining the actual value of $\chi(G)$ for the mesh in Figure (4.4) can be quite a challenge [25]. Never-

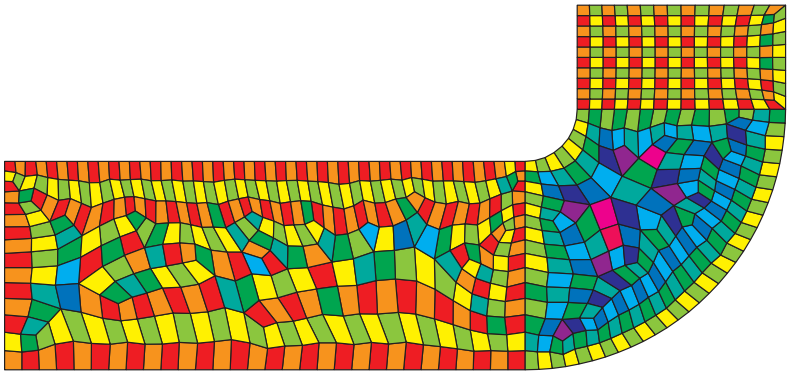


Figure 4.4: Unstructured mesh colored by the presented algorithm (708 elements, 128 threads and 12 colors).

theless, we can easily obtain a lower bound for the chromatic number based on two quantities (one from connectivity, and the other from the restriction): We will require at least the same number of colors as the maximum number of adjacent elements for a node, *or* the number of elements divided by the maximum number of available threads.

$$\chi(G) \geq \chi(G)_{LB} = \max\left[\max\left(elements/node_i\right), \frac{\#Elements}{\#Threads}\right] \tag{4.4}$$

The maximum number of elements adjacent to a node is a mesh constant (assuming the mesh does not change). The number of elements in the mesh is also constant. The only quantity changing is the number of threads. Given the number of threads we can compute the lower bound of the chromatic number, and use it to compare the efficiency of the coloring algorithm by knowing that $\chi(G) \geq \chi(G)_{LB}$.

Table 4.1: Number of colors used by the proposed algorithm compared to the lower bound for the chromatic number for the mesh in Figure (4.4).

Max. Threads	Algorithm Colors	$\chi(G)_{LB}$
256	8	5
192	8	5
128	12	5
96	13	6
64	16	12
48	20	15
32	25	23
24	34	30
16	47	45
12	62	60
8	91	89
4	178	178

From the data in Table (4.1), it is clear that as the mesh becomes larger (similar to reducing the number

of threads), the restriction on the max amount of threads kicks in and dictates over the optimal solution. For these cases the solutions obtained from the proposed algorithm and the optimal solution (bounded by $\chi(G)_{LB}$) are almost the same, that is, as the mesh size increases or the number of threads decreases, the algorithm becomes more efficient (Figure (4.5)).

The maximum number of elements per color is dictated by hardware limitations. For the currently available NVIDIA cores, the maximum limit is 512 threads per block, where all threads in the block are executed concurrently (In reality only a few threads are really executed concurrently in what is called a *warp*, but the user has no control over the warp. Hence it should be assumed that they all execute concurrently).

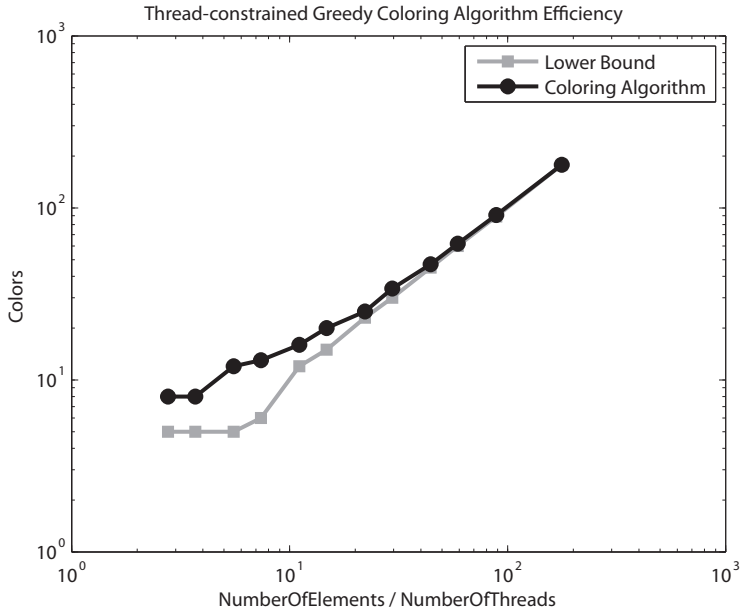


Figure 4.5: Analysis of the thread-constrained coloring algorithm efficiency relative to mesh size and number of threads.

4.3 Local Stiffness Matrices

In the element wise approach, each thread computes a local stiffness matrix for a given element at a specific time step (or more correctly said, a *stride*). This computation requires a lot of local (temporal) memory. Matrices that are used in the computation of $[k]$ are $[B]$, $[D]$, $[J]$, $[\Gamma]$, coordinates $[XY]$ among others. Because the current GPU architecture does not support a large number of registers (local thread memory), the code must be very efficient and store only what is strictly necessary during computation in order, to maximize the number of concurrent threads that we are able to launch.

For the case of a Q4 element, the size of the local stiffness matrix $[k]$ is 8×8. Thanks to symmetry, we

can store only half of it. We chose to work with the lower triangular matrix, and instead of storing all 64 elements, we only store 36. Even thought 36 numbers might not seem like much, one should keep in mind that this is per thread, and each thread will require more than just that to store other type of data during computations. If we only consider the storage of the local stiffness matrix for now, and assume it is completely stored in shared memory for illustration purposes:

$$[k^e]_{MEM\,req} = 36\frac{floats}{thread} \cdot 4\frac{bytes}{float} = 144\frac{bytes}{thread} \tag{4.5}$$

The total shared memory available is 16384 bytes, and if each thread uses 144 bytes, we could only launch 113 threads. Because of this, the local stiffness matrix will be split and stored half in shared memory, and half in registers as shown in Figure (4.6), allowing to launch more threads. By splitting the local stiffness matrix in two, the shared memory requirement per thread gets reduced to only 60 bytes (from the original 144 derived in Equation (4.5)), this raises the limit to 273 threads. In addition to being able to launch more threads, using registers increases the performance as well since local memory (registers) is faster than shared memory.

In addition to the registers used to store the local stiffness matrix, the assembly kernel requires a lot more

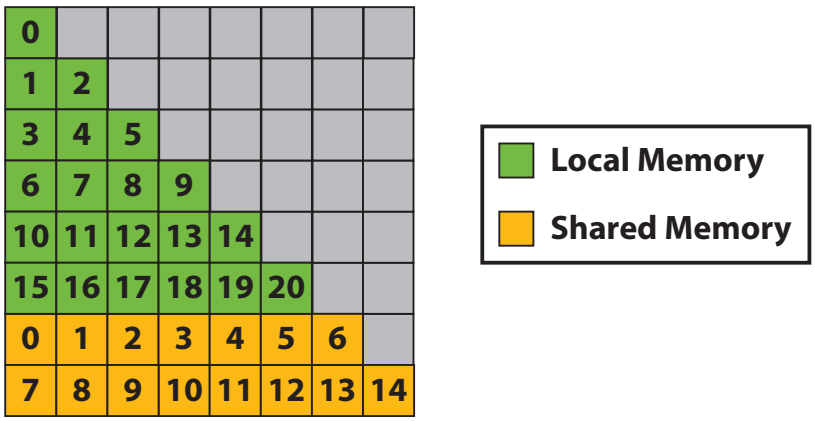


Figure 4.6: Memory allocation for the local stiffness matrix k^e for each thread.

registers for operations, loops, intermediate variables and such. The code was specially crafted to minimize the use of registers, nevertheless, the number of registers used by this kernel is 58. Considering that the total number of registers per block is 16384, the maximum number of threads according to this should be 282.

The number of threads that are finally launched for the stiffness matrix assembly is 256, this meets the requirements from all the possible limiting factors (hardware and memory limitations).

4.4 K Assembly Implementation

The global stiffness matrix $[K]$ gets completely assembled in *NumberOfColors* strides, obtained from the graph coloring scheme. For this data to be loaded into the GPU, a vector of length *NumberOfColors* $\times$ *NumberOfThreads* called *BlockElementList* is generated, where each block of *NumberOfThreads* tells the corresponding thread what element to integrate at that specific stride. Whenever the thread encounters a defined constant called *ElementsLimit*, it is basically instructed to wait and do nothing for that stride. An example for this *BlockElementList* is generated for the colored mesh in Figure (4.7). For this example, if $NumberOfThreads = 5$, and the defined value to represent no element is $ElementsLimit = 65535$. The vector that is sent to the GPU will have 5 blocks (5 colors) of 5 elements each (5 threads), that results in a total length of 25:

$$BlockElementList = \begin{bmatrix} 0 & 2 & 8 & 10 & 65535 \\ 11 & 12 & 1 & 3 & 65535 \\ 4 & 6 & 13 & 15 & 65535 \\ 5 & 7 & 14 & 65535 & 65535 \\ 9 & 65535 & 65535 & 65535 & 65535 \end{bmatrix}$$

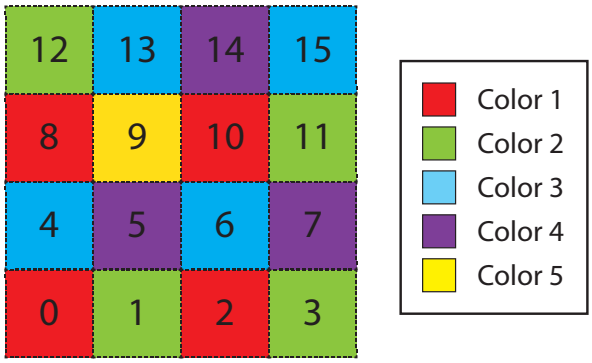


Figure 4.7: Example for generating the *BlockElementList* vector from a colored mesh.

Note that the fifth thread is idle for all strides, and the fourth thread only works for the first 3 strides. A simplified version of the assembly code is in Pseudo-code (4), here the variable *threadIdx.x* denotes the specific thread number. This code is too simple, and differs from the actual one quite a bit. The biggest differences are the data storage, where values known to be zero are neither stored nor allocated, and for symmetric matrices only the lower half is stored. In addition, most loops are unrolled to maximize speed and minimize the use of registers. In addition, because the algorithm only stores the lower half of the global

stiffness matrix, a check must be made to ensure assembly on the lower half only and never attempt to access a matrix element in the upper non-existent half.

Pseudo-code 4 Massively parallel $[K]$ assembly for Q4 elements based on graph coloring.

```
for (stride=0; stride<NumberOfColors; ++i) {
  MyElement = BlockElementList[stride*NumberOfThreads+threadIdx.x];

  % Check wether I have to integrate this element or not
  if (MyElement != ElementsLimit) {
    Get [D] based on Density[MyElement]
    Gather element coordinates XY for MyElement
    fill(Klocal,0);
    for(GaussPoint=0; GaussPoint<4; ++GaussPoint) {
      Compute [J] based on current GaussPoint
      Compute [B] based on [J]
      Klocal += [B']*[D]*[B]*J^(-1);
    }
    Assemble Klocal in Kglobal
  }

  % Make sure all threads reach this point before moving to a new stride
  __synchthreads();
}
```

4.5 Remarks

The scheme presented here effectively avoids race conditions in the stiffness matrix assembly. Some more efficient and sophisticated algorithms have been developed where not only the race condition is avoided [8], but also memory access is optimized therefore achieving a slightly higher speedup (but at the cost of a bigger overhead).

For the whole program to be efficient, the total runtime including the coloring or equivalent algorithm, has to be small. The greedy coloring algorithm is a bad coloring algorithm [24], but thanks to the maximum threads restriction (color size limitation), the algorithm has excellent results for large problems. For the case of topology optimization, the assembly process is done several times (once for each iteration). The coloring algorithm runtime is similar to the time required for one of these iterations, thus making the coloring algorithm application feasible. Computing the local stiffness matrices $[k]$ at every iteration is not really required. Inspecting Equation (2.53) it is clear that $[k_{e0}]$ does not change with ρ_e. This means that $[k_{e0}] = [k_e\,(\rho_e = 1)]$ can be computed only once, and stored to be used at every iteration premultiplying by the appropiate ρ_e^p before assembling. This approach would require some additional storage space (to store all the $[k_e]$ it is required 36 $bytes \cdot NumberOfElements$) and it is not compatible with a CAMD approach (future developments).

The race condition for the force vector $\{F\}$ can be avoided if we only consider displacement and force

boundary conditions. If that is the case, and if the mesh does not change between iterations, then the force vector will not change either. This means that the force vector can be assembled once in the CPU (sequentially), stored, and used for each iteration.

The assembly process speed will depend on the results from the coloring algorithm (that depends itself on the specific mesh to be colored), but speedups of 6 to 8 compared to the CPU are typically observed (Dual-Core AMD Opteron^{TM}Processor 2216 and a Tesla T10 Processor). The kernel that computes the sensitivities and compliance for the problem shares a large amount of code with the assembly kernel, and falls into the same hardware and memory limitations, because of that, the sensitivity kernel is also limited to launch 256 threads, whereas all the remaining kernels launch the maximum amount of threads possible, that is 512.

Chapter 5

Sensitivity Filtering

The topology optimization problem (material optimization) is continuous in nature. Instead we are using a finite discrete formulation, which give rise to some artificial anomalies in the design. The most important ones are the *checkerboard pattern* and the *mesh dependence* of the solution. The checkerboard pattern was explained previously (Figure (2.8(a))). The mesh dependence, as the name states, means that the final solution is dependent on the specific mesh. Both problems can be eliminated or greatly reduced using a sensitivity filter [54, 42].

The sensitivity filter acts by blurring the sensitivity distribution throughout the mesh, effectively eliminating the checkerboard pattern. A side effect of this blurring is the widening of what would otherwise be thin elements or bars in the topology. The sensitivity filter does not directly impose a member size constraint, but its effect is similar. If the mesh gets modified, but the filter parameters are kept the same, the distribution of the sensitivities after filtering should resemble those in the old mesh, and the member locations and sizes should remain approximately the same or exhibit little change.

5.1 Density Variable Location

The current implementation of the code uses Q4 elements, with a single material density variable per element. So far, we were not required to know the exact location of this variable (only had to know the element it belonged to). In order to apply the filter to a specific element, we need to know the distance from this element's variable to the material density variable of other elements. For the case of a structured mesh, the location of this point is obviously located at the center of the square or rectangle.

The most obvious location for the material density variable is the centroid of the element. Nevertheless, we can recognize two centroids for a quadrilateral of irregular shape, resulting in slightly different points (Figure (5.1)). One centroid results from computing the center of mass (center of gravity) and the other is the geometrical centroid, obtained strictly from geometrical constructions.

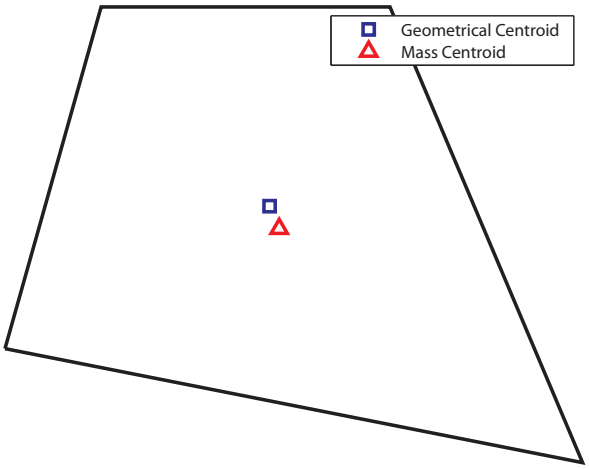
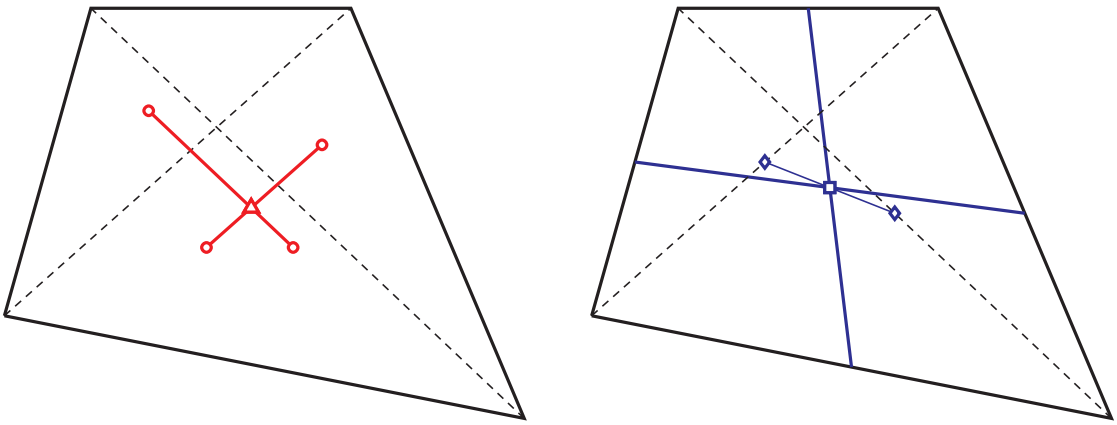


Figure 5.1: Two possible options for the location of the material density variable.



(a) Construction for the center of mass.

(b) Construction for the geometrical centroid.

Figure 5.2: Constructions for the center of mass and geometrical centroid of a quadrilateral.

The center of mass can be obtained dividing the quadrilateral in two triangles, computing the center of gravity of the triangles (average of vertices) and from that obtaining the center of mass of the quadrilateral as a weighted average:

$$C_M = \left\{ \frac{A_1 \cdot C_1^x + A_2 \cdot C_2^x}{A_1 + A_2} \quad , \quad \frac{A_1 \cdot C_1^y + A_2 \cdot C_2^y}{A_1 + A_2} \right\} \tag{5.1}$$

where A_1 and A_2 denote the areas of both triangles, the centers of mass for each triangle are $\{C_1^x, C_1^y\}$ and $\{C_2^x, C_2^y\}$. Instead of doing the weighted average, a slightly more complicated, but more graphical way to get this point is to draw a line between the centers of mass of both triangles, and do the same procedure

subdividing the quadrilateral with the other diagonal. The intersection of both lines corresponds to the center of mass of the quadrilateral (Figure (5.1)).

The geometrical centroid is easier to obtain, and several different geometrical constructions can be made. The quickest and most straight forward is to average all 4 vertices:

$$C_G = \left\{ \frac{X_1+X_2+X_3+X_4}{4} \quad , \quad \frac{Y_1+Y_2+Y_3+Y_4}{4} \right\} \tag{5.2}$$

Other options to find this point include the intersection of the medians, or the midpoint of a line drawn between the midpoints of the diagonals (Figure (5.2)). If we compare both centroids for a quadrilateral with vertices $P_1 = \{0, 0\}$, $P_2 = \{6, -1\}$, $P_3 = \{4, 3\}$ and $P_4 = \{1, 3\}$, it is obvious that there will be a difference between both. For this specific case, the centroids are:

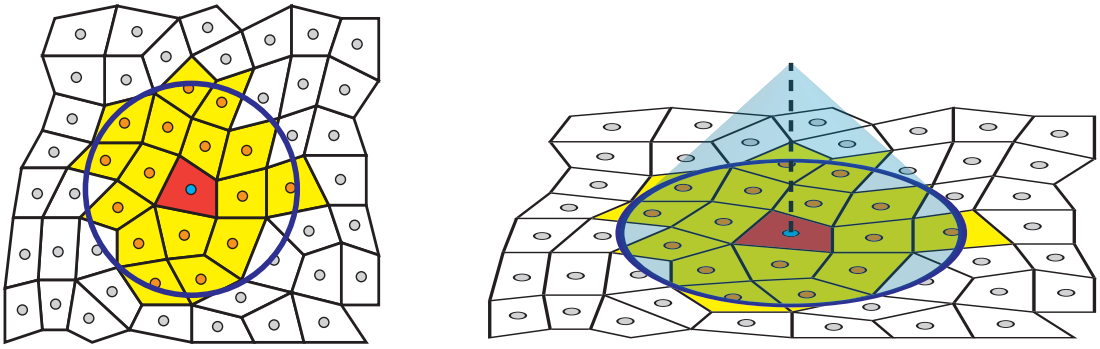
$$C_G = \left\{ 2.75 \quad , \quad 1.25 \right\}$$
$$C_M = \left\{ 2.85 \quad , \quad 1.05 \right\}$$

The geometrical centroid (Equation (5.2)) generally lacks the nice physical properties that the actual center of mass has (Equation (5.1)), but computing the geometrical centroid is trivial compared to the other. Because of easiness and simplicity of the resulting code, the material density variable will be located at the geometrical centroid. Nevertheless, the difference between both of them is minimal for close-to-regular shaped elements, and differences in the topology design range from small to none when using one or the other.

5.2 Filter Search

The sensitivity of each element is averaged in some way with the sensitivities of the neighbors that fall within the projection of the convolution function, as in Figure (5.3(a)). The convolution function typically used is linearly decaying to 0, that is equivalent to a cone (Figure (5.3(b))), although other types of convolution functions can also be used. There is a problem with unstructured meshes because it is computationally expensive to find the elements that fall within an element's filter. A somewhat efficient scheme has to be devised to compute and store this information. Ideally, we would not like to compute this information at each topology optimization iteration.

We want to find all the neighboring elements within the radius of the filter for each element. A brute force approach is unfeasible for large meshes, but using the information from the communication matrix



(a) Filter radius indicating the neigbours that fall within this element's filter.

(b) Convolution function (cone).

Figure 5.3: Filtering of sensitivities.

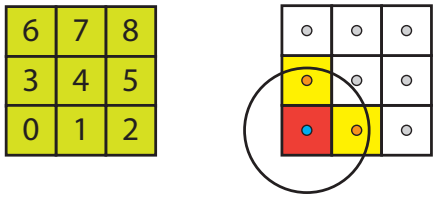
(communication graph), a somewhat efficient search algorithm can be devised.

The basic idea of the algorithm is to find the direct neighbors of the element, and evaluate if they are within the filter circle. If an element is within the circle, then this element's neighbors are tested as well. The neighbors of each element get recursive test until the neighbors of the next element are outside the filter. To that effect, we keep track of what elements still need to be checked in either a queue of a stack. The difference between them is that the queue works as a FIFO data structure (First-in-First-out), whereas the stack operates in a LIFO way (Last-in-First-out).

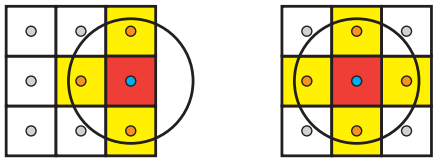
The consequence of using a FIFO or a LIFO list of elements to be evaluated by the filter results in either a *Breadth-first search* (BFS) for FIFO or *Depth-first search* (DFS) for LIFO. In BFS, the algorithm explores all the closer elements before moving to a new depth level in the graph (connectivity graph), whereas in DFS, the algorithm explores the graph in full depth, before moving to a new neighbor (next branch in the graph) of the initial element (starting point of the graph). Both algorithms have very similar performance, and both can suffer from worst case scenarios depending on the specific graph to be explored. There is no significant difference between both approaches, and in the proposed algorithm BFS was chosen for the search.

5.3 Filter List Construction and Storage

The algorithm does not have a priori information regarding how many elements fall within each element's filter. Therefore, for each element, there is an unknown list of elements to store. Because the number of elements depends on the specific mesh, we need a dynamic array of dynamic lists (one list per element in the



(a) Element numbering. (b) Corner element filter.



(c) Edge element filter. (d) Inner element filter.

Figure 5.4: Different filter projection situations for an adimensional structured 3×3 mesh with $R = 1.25$.

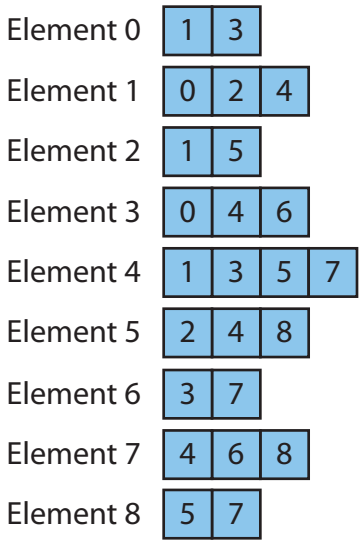


Figure 5.5: Filter lists for an adimensional structured 3×3 mesh with $R = 1.25$.

mesh). This is a problem even for structured meshes, if one considers that elements in the edge or corner have a different number of neighbors that fall in the filter compared to an inner element. An example of this situation for a simple structured 3×3 mesh with the filter radius set to be 1.25 times the element size is shown in Figure (5.4), where the final filter list for each element is given in Figure (5.5).

A widely accepted implementation for dynamic structures in C++ such as linked lists, doubly linked lists and others is the *Standard Template Library* (STL) [82]. With the STL we want to generate *NumberOfElements* vectors. Once the algorithm computes all the filter lists, we would like to organize all the information in a way that is easy to transfer and read in the GPU. The proposed approach is to concatenate all the filter lists into one big 1D array, and have a secondary array with the indices (pointer address offset) to the heads and tails of each sequence of numbers. For the case presented in Figures (5.4) and (5.5), the final *FilterList* and *FilterListIndex* are:

$$FilterList = [1 \quad 3 \quad 0 \quad 2 \quad 4 \quad 1 \quad 5 \quad 0 \quad 4 \quad 6 \quad 1 \quad 3 \quad 5 \quad 7 \quad 2 \quad \hookleftarrow$$
$$4 \quad 8 \quad 3 \quad 7 \quad 4 \quad 6 \quad 8 \quad 5 \quad 7]$$
$$FilterListIndex = \begin{bmatrix} 0 & 2 & 5 & 7 & 10 & 14 & 17 & 19 & 22 & 24 \end{bmatrix}$$

The information stored in these two arrays have everything needed to allocate, copy and read information regarding the filter (sparse storage): the length of *FilterListIndex* is $NumberOfElements + 1$, the length of *FilterList* is $FilterListIndex[NumberOfElements]$, and the filter list corresponding to element i spans from $FilterList[FilterListIndex[i]]$ to $FilterList[FilterListIndex[i+1]]$. The length of the list is the index difference in positions i and $i+1$. The algorithm to construct the *FilterList* and *FilterListIndex* is presented in Pseudo-code (5). This code is executed only once in the CPU, and the information to be used at each iteration is stored and transferred to the GPU (or kept at the CPU if the CPU compute chain is selected).

Pseudo-code 5 FilterList construction algorithm with BFS

```
for (i=0; i<NumberOfElements; ++i) {
  fill(ElementsTested,false);

  % Kickstart search algorithm adding this element to the queue
  FilterQueue[0]=i;
  ElementsTested[i]=true;
  QueueLength=1;

  % Traverse the queue in a FIFO way
  for (j=0; j<QueueLength; ++j) {
    CurrentElement=FilterQueue[j];

    if ( Distance(i,CurrentElement)<FilterRadius) {
      for (k=0; k<CurrentElement.Neighbors; ++k) {
        % Has it been already added to the queue?
        if ( ElementsTested[CurrentElement.Neighbors[k]]==false ) {
          FilterQueue[QueueLength]=CurrentElement.Neighbors[k];
          ++QueueLength;
          ElementsTested[CurrentElement.Neighbors[k]]=true;
        }
      }
      % Add this element to the FilterList
      FilterList[i].push(CurrentElement);
    }

  } % Queue traverse loop
}% Elements traverse loop

Contatenate FilterLists in a 1D array
Construct FilterListIndex
```

5.4 Implementation

The filtering process in the GPU takes place in an element wise manner. That is, each thread will be responsible for the filtering of a single element at a time. If a single thread is in charge of applying the filter procedure to an element: this thread has to read the sensitivities of all elements within the element's filter projection, apply the convolution formula, and write the new filtered sensitivity. Because reading information is safe from a race condition point of view, and each thread writes the sensitivity of the element it is in charge with, there is no race condition and also no need for synchronization barriers. After loading

the *FilterList* and *FilterListIndex* in the GPU, the filter algorithm for a linear convolution function (cone) in accordance with Equation (2.66) is presented in Pseudo-code (6).

Pseudo-code 6 Parallel sensitivity filter code

```
for (i=0; i<Strides; ++i) {
  MyElement = threadIdx.x * Strides + i;

  if (MyElement < NumberOfElements) {
    Denom = FilterRadius;
    Numer = FilterRadius * Density[MyElement] * Area[MyElement] * Sensitivity[MyElement];

    for (j=FilterListIndex[MyElement]; j<FilterListIndex[MyElement+1]; ++j) {
      CurrentElement = FilterList[j];
      Denom += Distance(MyElement,CurrentElement);
      Numer += Distance(MyElement,CurrentElement) * Density[MyElement] * Area[MyElement] * Sensitivity[MyElement];
    }

    NewSensitivity[MyElement] = Numer / ( Denom * Density[MyElement] * Area[MyElement] );
  }
}
```

5.5 Remarks

The sensitivity filter algorithm is easy to adapt into a parallel code. The difficult part of the filter is computing the filter list for each element, problem that occurs even in sequential codes provided that we are dealing with an unstructured mesh. The current proposal is to compute this information beforehand and store it, so that it can be used at each iteration. The search has to be able to obtain the information desired without traversing the entire mesh every time the filter gets applied to an element. A brute force approach for the search would scale too rapidly with the mesh size making it unpractical for large problems. The filter algorithm does not suffer from race condition, at least not in this approach, and the code can concurrently process any number of elements at any given order. With this filter, the solutions for the final topology can be tailored towards more feasible designs, in addition to eliminating artificial structures such as checker boarding.

Chapter 6

Other Functions and Kernels

In the present implementation framework, problems are generated and saved into a standard format, e.g. the SIMULIA's Abaqus file format (*.inp extension). A parser reads the mesh, boundary conditions and material properties from this file. The topology optimization loop consists of several different functions executed in sequence. However, before the topology optimization loop begins, we need to compute some information beforehand, which is called the *precruncher*. The precruncher executes only once and it is not part of the topology optimization loop.

6.1 CPU Functions

The topology optimization loop is implemented in equivalent functions for both the CPU and GPU. Nevertheless, there are two modules (set of functions) that must be executed no matter what topology optimization chain is selected (CPU or GPU). These modules are always executed in the CPU[1], and provide the information required to perform the topology optimization loops.

6.1.1 Precruncher

The precruncher obtains essential information for the code to run. It makes no difference wether the CPU or GPU chain is selected, information from the precruncher will be required. The precruncher is always executed in CPU. The tasks done in the precruncher are:

- Build communications graph
- Graph coloring

[1]The function that computes the element and domain areas can be either executed in the CPU or GPU

- Calculate the bandwidth of $[K]$
- Calculate the elements and domain areas
- Build the filter list

The communication graph is used by both the graph colorer and the filter list construction. The graph coloring is required to assemble $[K]$ in a parallel way without falling into race conditions. The bandwidth of $[K]$ is required by the assembly process and the solver to make the memory usage of $[K]$ smaller, more efficient, and traversing the entire matrix efficiently. The element and domain areas are required by the filter in the convolution function and in the material update module to enforce the volume fraction constraint. The filter list is required by the filter module to avoid searching through the mesh at every iteration.

6.1.2 Input File Parser

The problems are read into the program via Abaqus [81] file format, although parsers for other formats can be easily written and integrated into the code. This scheme allows us to use advanced meshers such as MSC Patran [79] or Abaqus/CAE[2] itself, as well as the integrated bandwidth reduction algorithms already implemented in these. Since we are trying to take advantage of the flexibility offered by unstructured meshes, it makes sense to outsource the mesh creation to a specialized software.
The file is read without any modification from these third party codes, and the following data is obtained:

- Nodal locations
- Element nodes
- Material properties
- Boundary conditions

In its current implementation, the Abaqus parser code has full support for Q4 elements, and partial support for other type of elements and tridimensional problems as well. Currently it can only handle one material property and constitutive law per problem. Mixed element types or different properties throughout the mesh are not yet supported.
The main reason for having the Abaqus parser is the ability to create and read complex problems, but a second reason just as important is to use the bandwidth reduction algorithms available to maximize the speed of banded solvers.

[2]Abaqus/CAE is the pre and post processor of the Abaqus suite.

6.2 CUDA Kernels

In addition to the solver, assembly and filter kernels previously explained, other kernels complete the topology optimization loop in the GPU. The complete list of kernels is:

- Calculate element area[3]
- Stiffness matrix assembly
- Boundary Conditions application
- Solver[4]
- Sensitivity calculation
- Sensitivity filter
- Material Update

These complete the topology optimization loop and enables us to run the topology optimization algorithm entirely on the GPU (with the exception of the Abaqus parser and precruncher that are not really part of the loop).

6.2.1 Boundary Conditions

There are a couple of ways to approach the boundary conditions problem. Forces and boundary tractions are somewhat easy to apply since they only add to the force vector on the right hand side. Displacements boundary conditions on the other hand are trickier in the sense that in most cases the stiffness matrix has to be modified (either reduced in size or values in it must change).

The imposed displacement boundary condition is a known quantity (a fixed boundary condition is a special case of an zero imposed displacement), we can then move these to the right-hand-side of the equation (they are not unknowns). Using index notation, if the displacement imposed degrees of freedom are denoted by index d, with imposed displacements u_d, and the free degrees of freedom are all of those specified by indices

[3]The element area kernel is not part of the topology optimization loop, but can be executed only once at the precruncher level to obtain the element areas and total domain area. There is a CPU and GPU version of the element and domain area functions.

[4]The solver kernel is in fact the BaCh solver, that can be replaced by the CPU equivalent in LAPACK.

f, the formulation is found to be:

$$
\begin{aligned}
[K]\{u\} &= \{F\} \\
\begin{bmatrix} K_{ff} & K_{fd} \\ K_{df} & K_{dd} \end{bmatrix} \begin{Bmatrix} u_f \\ u_d \end{Bmatrix} &= \begin{Bmatrix} F \\ R \end{Bmatrix} \\
[K_{ff}]\{u_f\} &= \{F\} - [K_{fd}]\{u_d\}
\end{aligned} \tag{6.1}
$$

where $\{R\} = [K_{df}]\{u_f\} + [K_{dd}]\{u_d\}$ are the reactions (forces) at the displacement imposed points that can be obtained after the system is solved. The force vector for the system with the applied boundary conditions is $\{F^*\} = \{F\} - K_{fd} \cdot \{u_d\}$. Note that if all the displacement boundary conditions are fixed, that is $\{u_d\} = \{0\}$, then $\{F^*\} = \{F\}$. The actual system to be solved has a smaller sized matrix (the size is $f \times f$ to be more precise, as seen in Equation (6.1)).

The typical approach is to assemble the complete $[K]$ and later resize it to only $[K_{ff}]$ before solving the system. The additional information contained in $[K]$ (like $[K_{fd}]$) is used to obtain the modified force vector $\{F^*\}$. The problem with this approach, is that resizing the already allocated computer memory is not possible. Unless advanced data structures are used (like *linked lists*, *doubly-linked list* and others[5]), what happens in reality is that a new memory space is allocated and only the desired information is copied to this new memory (hence a resize).

A second more elaborate approach, consists in checking every degree of freedom before it is assembled into $[K]$, to see if it is displacement constrained. If that is the case, the value is not stored in $[K]$ but rather used to alter $\{F\}$ instead. The downside of this approach is that several checks have to be made before assembling a single value in $[K]$. A more efficient variation of this approach is to prepare an array of mappings for the degrees of freedom, where the free ones map into positions of the actual $[K_{ff}]$, while the constrained ones map to $[K_{df}]$, $[K_{dd}]$ and $[K_{fd}]$. This scheme is very efficient, but requires a conditional evaluation before assembling (although it is cheap), and in addition, the mapping to be calculated beforehand.

A third approach, with an overall performance between the previous two proposals is to do a mixed formulation. This means that the right hand side vector $\{b\}$ in $[A]\{x\} = \{b\}$ has forces and displacements in different positions. The biggest disadvantage of this formulation is that the system that is solved has the original size of $[K]$, translating into more operations. The additional computational cost of this third

[5]These special data structures and others are available in the Standard Template Library (STL). Most of them provide resizing functionality, but at the cost of performance.

approach gets compensated by the simplicity of the algorithm. The formulation becomes then:

$$\begin{bmatrix} K_{ff} & K_{fd} \\ 0 & I \end{bmatrix} \begin{Bmatrix} u_f \\ u_d \end{Bmatrix} = \begin{Bmatrix} F \\ D \end{Bmatrix} \tag{6.2}$$

where [0] denotes a zero matrix, $[I]$ is the identity matrix and $\{D\}$ are the prescribed displacements. Viewed in matrix form, the rows corresponding to prescribed displacement boundary conditions are changed to all zeroes except for the diagonal that has a "1" on it. In addition to this, the row in the force vector has to be changed to the prescribed displacement as portrayed in Figure (6.1).

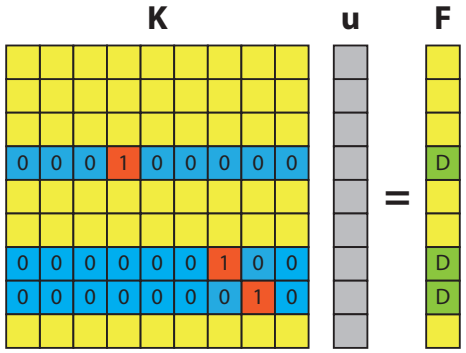


Figure 6.1: Modification for displacement BC (non-symmetric).

This approach is valid, but has no symmetry in the stiffness matrix, property we require to use the a Cholesky solution scheme. If we restrict the imposed displacement boundary conditions to only be of fixed type $\{D\} = \{0\}$, then the Equation (6.2) can regain its symmetry:

$$\begin{bmatrix} K_{ff} & 0 \\ 0 & I \end{bmatrix} \begin{Bmatrix} u_f \\ u_d \end{Bmatrix} = \begin{Bmatrix} F \\ 0 \end{Bmatrix} \tag{6.3}$$

The system in Equation (6.3) is indeed symmetric and has fixed boundary conditions. The rows and columns corresponding to a fixed degree of freedom are zeroed as shown in Figure (6.2).

If each thread is in charge of zeroing a row and column, then there will be some conflicts at some positions, where two threads will attempt to write a zero at the same time. But there is no race condition danger since both threads are trying to write the same value (a zero).

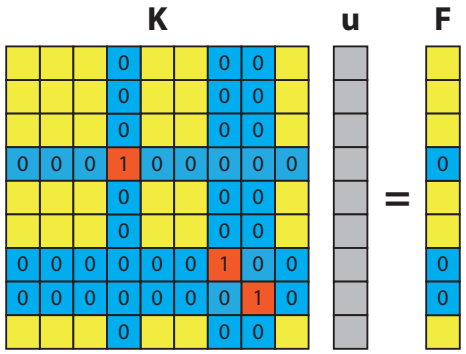


Figure 6.2: Symmetric modification for fixed displacement.

6.2.2 Sensitivity Calculation

The sensitivity kernel is almost the same as the assembly kernel, with two main exceptions. The first one being the fact that since there is no assembly taking place, hence the kernel has no race condition problems. The second difference is that in order to compute the compliance, the kernel has to sum compliances from all elements.

The compliance of each element is computed and added to a local variable within each thread. Then all the threads copy their value of compliance to a shared memory, where a binary reduction is carried out. During the reduction each thread reads its corresponding position in the shared memory array, and reduces it with the same position shifted by a stride. In our case we want the compliance, and therefore the reduction is an addition operation. With each stride, the number of threads that operate on data gets reduced by half. A graphical representation of the process for only 16 elements is illustrated in Figure (6.3).

Because this code has similar memory requirements as the assembly kernel, the maximum number of threads is also limited and set to be 192.

The number of required strides to reduce all data is:

$$\begin{aligned} 2^{strides} &\geq NumberOfThreads \\ strides &= \text{ceil}\left(\log_2 NumberOfThreads\right) \end{aligned} \tag{6.4}$$

where ceil () is the round-up or ceiling function. A basic code for the reduction is available in Pseudo-code (7).

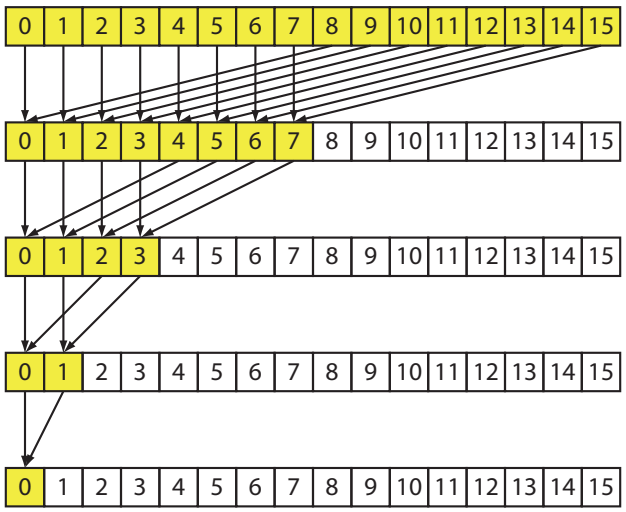


Figure 6.3: Binary reduction scheme with minimal memory bank conflicts.

Pseudo-code 7 Array reduction algorithm with minimal bank conflicts

```
for (i=stride; i>=0; --i) {
  offset=pow(2,i);
  if (threadIdx.x < offset) {
    SharedArray[threadIdx.x] += SharedArray[threadIdx.x+offset];
  }
}
```

6.2.3 Material Update

The material update is done by finding the Lagrange multiplier (λ) from the topology optimization formulation that optimizes the objective function subjected to the volume constraint (Equations (2.55) and (2.62)). In the search for this λ, typically a binary partition is done. But by using several parallel threads, we can subdivide the search domain into more than 2 pieces. Using the maximum number of threads available per block, we can subdivide the search domain in 512 for every iteration.

Each thread takes a value for λ from a linear partition of the search domain, and computes the total material volume for that given value. Each thread stores this volume in a shared memory array. The range that has the correct value of λ is such that the previous thread obtains a higher than specified volume fraction (smaller than required λ), and the following thread has a smaller than specified volume fraction (larger than required λ). To find this, each thread reads its own value and the value for the thread right after (Figure (6.4)). The thread with two values bounding the volume constraint, modifies the search domain to be this specific smaller partition and the partition process if repeated. After a couple of iterations, the value of λ gets narrowed down to a very small range, and the average is taken to be the actual value for λ.

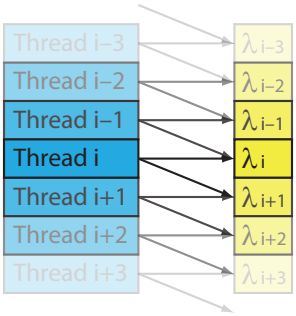


Figure 6.4: Parallel search for λ.

The search domain is initially specified by the range $[\lambda_{\max}, \lambda_{\min}]$ that represents the maximum and minimum values that λ can take. In other words $\lambda_{\min} \leq \lambda \leq \lambda_{\max}$. Since we are using a massively parallel computer, these partitions are extremely efficient at narrowing down the actual value of λ. The stopping criteria for the search is therefore a fixed number of iterations instead of a specified tolerance. Nevertheless, there is still control over the desired precision for λ. This precision can be specified by a prescribed number of partition iterations as:

$$\begin{aligned} \varepsilon &= \frac{\lambda_{\max} - \lambda_{\min}}{512^n} = \frac{\lambda_{\max} - \lambda_{\min}}{2^{9n}} \\ n &= \frac{1}{9} \log_2 \left(\frac{\lambda_{\max} - \lambda_{\min}}{\varepsilon} \right) \end{aligned} \tag{6.5}$$

Given the desired precision ε, the number of iterations required n can be determined from Equation (6.5). A very simplified implementation of the parallel search is in Pseudo-code (8).

The topology optimization iterations will continue until either one of two user defined stopping criteria are met: A specified number of iterations or a tolerance in the change variable. Because of the different operating precision of the CPU and GPU, as well as different solvers available, the convergence path the code takes will be different in each case. Therefore, for comparison purposes it is better to specify the number of iterations, and no tolerance. In actual applications the opposite is generally desired, and a change tolerance is specified (to prevent the code from hanging if convergence is never met, the maximum number of iterations can also be used as well).

The change variable as defined in Equation (2.70) is valid for most applications. But with unstructured meshes, there is the phenomena of material oscillation between two very small elements (an unstructured mesh will typically have elements of varied sizes), and the fact that we are using single precision arithmetic makes this change variable behave erratically or oscillating (the change variable as defined by Equation (2.69) is even worse in this matter). It would be better to have a more stable variable to monitor since there

Pseudo-code 8 Optimality Criteria parallel search algorithm for λ

```
if (threadIdx.x==0) { %Define the range of the domain
  SharedDomainBegin = LambdaMin;
  SharedDomainRange = LambdaMax - LambdaMin;
}

for (i=0; i<Iterations; ++i) { %Iterate n times to desired precision
  __synchthreads();
  MyLambda = SharedDomainBegin + SharedDomainRange * threadIdx.x / NumberOfThreads;
  Calculate Be
  Calculate MyVolume for MyLambda
  SharedVolumes[threadIdx.x] = MyVolume;
  __synchthreads();
  if (SharedVolume[threadIdx.x]>VolumeFraction && SharedVolume[threadIdx.x+1]<VolumeFraction) {
    SharedDomainBegin = MyLambda;
    SharedDomainRange /= NumberOfThreads;
  }
}

__synchthreads();
if (threadIdx.x==0) { %The final value for lambda is...
  Lambda = SharedDomainBegin + 0.5 * SharedDomainRange;
}
```

is no real interest in these very small material oscillations. A refinement of the change variable that includes the element volume is:

$$change = \frac{\sum_{e=1}^{n} \|\rho_e^{new} - \rho_e\| \cdot V_e}{V} \tag{6.6}$$

The expression for the change variable in Equation (6.6) does not get affected by the small element oscillations by taking into account the element sizes. This results in a very stable decreasing behavior of the change variable, that will also range from 0 to 1 (the maximum will actually depend on the moving limit variable), making it excellent for the stopping criteria.

6.2.4 Element Areas

The element area for a Q4 with counter-clockwise numbering is::

$$A = \frac{1}{2}\left[(x_1 - x_3)(y_2 - y_4) + (x_2 - x_4)(y_3 - y_1)\right] \tag{6.7}$$

and the total domain area can be computed as a reduction just like in Figure (6.3) and Pseudo-code (7). Some NVIDIA architectures do not have de-normalized numbers, and considering that we are operating with single precision, the sum of all areas in a sequential code can easily have rounding errors due to additions of very big and very small numbers. In the parallel scheme, the numbers being added are of similar magnitude, thus reducing these errors (each stride adds two similar magnitude values).

6.3 Remarks

There are several different steps involved in the topology optimization algorithm. Most of these steps are easy to parallelize, nevertheless, some decisions taken in these implementations add restrictions or limitations to the code in favor of simplicity or speed. Some of the of the decisions made are the inability of the current code to handle displacement boundary conditions other than fixed, or specifying a priori the precision in the Lagrange multiplier based on a fixed number of subdivisions.

The input was chosen to be the Abaqus file format because of the flexibility of software that support Abaqus files (MSC Patran and Abaqus/CAE itself among others). Implementation is only available for Q4 elements in the current state of the code. The code could be easily modified in the future to support more input file formats, or to produce Abaqus-like output, enabling the code to be used as a FEM solver much like Abaqus or MSC Nastran.

Chapter 7

Examples & Benchmarks

The performance and results for all the different compute chain options are compared using 3 emblematic and/or interesting problems:

1. Bike frame
2. Messerschmitt-Bölkow-Blohm (MBB) beam
3. MBB beam with holes

The bike frame conceptual design is a somewhat applied problem, where small differences in the final design can be attributed to the specific selection of compute chain and solver. The MBB beam is a typical problem solved in topology optimization, and it is of particular interest since an undesirable situation takes place where the algorithm falls into a local minima depending on the solver used. The MBB beam with holes is a variation of the typical MBB problem, where a unstructured mesh is required to properly represent the domain restriction.

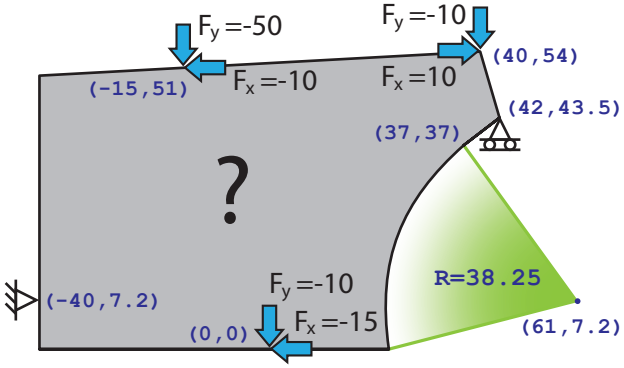
In addition to the analysis of the final designs, the code is benchmarked and compared against the CPU chain, and speedup against this is measured for all possible options. If the entire topology optimization compute chain is in the CPU, then the results are tagged as *CPU*, and the same applies for the GPU. If the results are obtained from the CPU chain, but using a GPU solver, then the chain is called a *CPU hybrid*, or CPU chain with an e**x**changed solver, and the results are tagged as *CPU -X*. For the case of a GPU chain with a CPU solver, the results are tagged as *GPU -X*. If the bandwidth (b_w) is bigger than 513, then only two compute chains are available: CPU and GPU -X (BaCh solver cannot handle a bandwidth bigger than 513). A picture of all the available compute chains is available in Figure (1.3). The penalization for all three problems is $p = 3$, and they are all run to 30 iterations.
The total runtime is used to compute the speedup over the CPU code, with the runtime also broken into its 3 major components: Input file parsing, precrunching and topology optimization loop.

The hardware used for all benchmarks consists of a dual-socket dual-core AMD Opteron 2216, 8GB of RAM and a NVIDIA Tesla T10 GPU with 4GB of RAM.

7.1 Bike Frame

A bike frame is a common structure, that continuously seeks for lighter and stronger designs. A simplified problem based on real bike frame geometries was created. Loads are artificial but are expected to resemble a real loading scenario that might occur while using the bike. The domain has to accommodate several restrictions. The clear space for the front wheel to turn sideways, the location of the topmost bar, the total span of the frame are some of the restrictions taken into account while defining this domain. Following a real life bike (Figure (7.1(a))), the loads and the domain are described in Figure (7.1(b)), with all units in centimeters and kilograms. The mesh has 20378 elements, 20635 nodes and the resulting stiffness matrix

(a) Mid-high end bike available in the market (2010 Cannondale Capo 2 urban commuter bike. Extracted from [87]).



(b) Bike domain, loads and boundary conditions.

Figure 7.1: Bike frame model and real design analogy.

bandwidth is 350. The mesh is available in Figure (7.2). The material is taken as Aluminum with a unit depth, elastic modulus $E = 700000kg/cm^2$, and $\nu = 0.3$.

Topologies obtained with each one of these compute chains for a volume fraction of $volfrac = 0.3$, filter radius $R = 1.2$ and 30 iterations are shown in Figure (7.3). The different machine precision, floating point standards, and orders of operations make for slight differences in the final design, but the overall concept is similar and matches what is expected for a bike frame. Figure (7.4) shows the result from the hybrid GPU chain (GPU -X), with the remaining bike components traced in dashed lines. Analysis of the change variable and the compliance show a very stable and steady decrease for all cases (Figure (7.5)).

The total runtime for all options is in Figure (7.6(a)) and the speedup in Figure (7.6(b)). The speedup

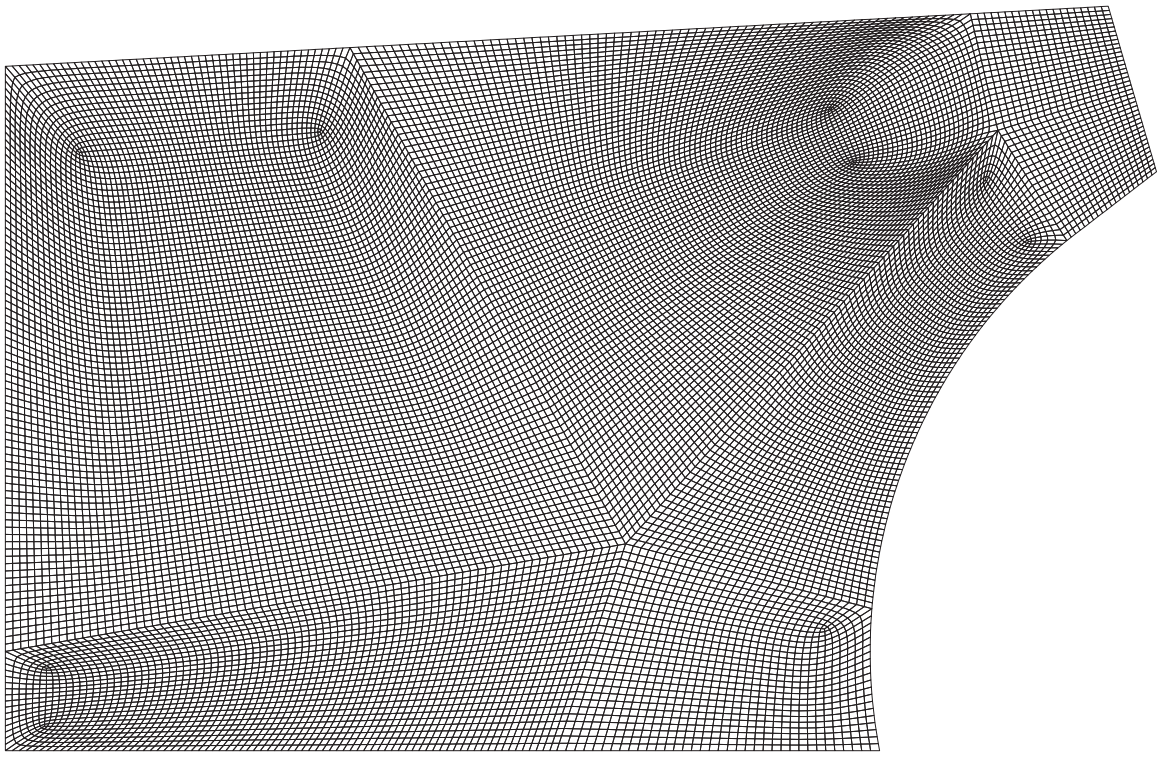


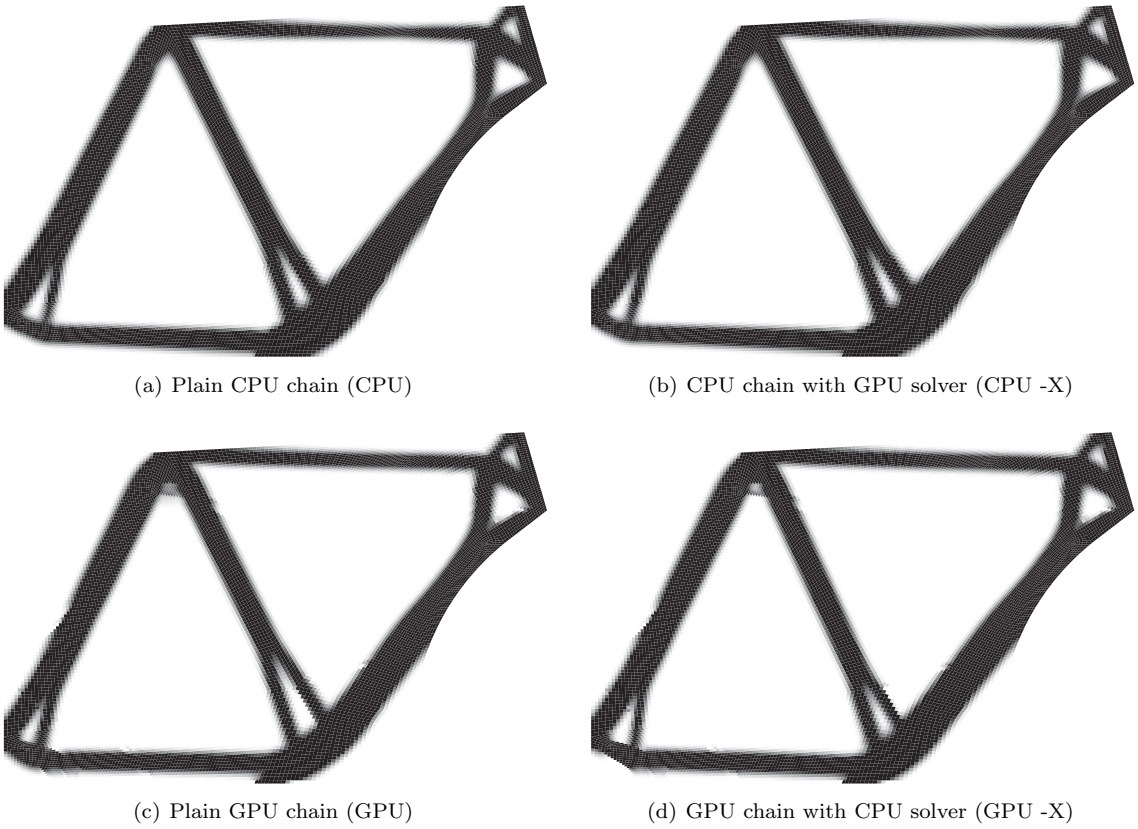
Figure 7.2: Bike frame mesh (20378 elements, 20635 nodes).

of the plain GPU code is close to 0.25, meaning that the overall runtime is 4 times slower than the CPU version of the code. The hybrid GPU chain with CPU solver has a speedup of almost 1.1. This is interesting because this compute chain has to transfer the information from the GPU to the CPU every time the system is to be solved, and then transfer the results back. This is additional work that the CPU chain does not have to do. This means that the hybrid GPU topology optimization chain is much faster than the CPU one, but some of this advantage is probably lost in the memory transfer.

7.2 Messerschmitt-Bölkow-Blohm (MBB) Beam

The MBB beam is a typical example in topology optimization. In addition to that, this mesh is structured. The code makes no difference between structured or unstructured meshes. The domain, loading and boundary conditions were explained in Figure (2.5). The mesh has 43200 elements, 46381 nodes and a bandwidth of 486. The mesh can be seen in Figure (7.7).

The topologies were obtained for a volume fraction $volfrac = 0.3$, filter radius $R = 0.02$ and 20 iterations. Results obtained for all compute chains are available in Figure (7.8). The solutions for the GPU solver



(a) Plain CPU chain (CPU)

(b) CPU chain with GPU solver (CPU -X)

(c) Plain GPU chain (GPU)

(d) GPU chain with CPU solver (GPU -X)

Figure 7.3: Bike frame results for all 4 compute chains after 30 iterations each.

(BaCh solver) fall into a local minima. The trend towards this local minima is appreciated from the very first iterations. Figure (7.9) show the topologies after only 2 iterations. It is clear that for the GPU solver (Figures (7.9(b)) and (7.9(c))), the typical MBB solution is still somewhat visible (in a very faded gray) behind the massive material accumulation on the left, but it completely disappears towards the end.

The compliance and change variable for all problems decrease monotonically (Figure (7.10)), except that for the cases that fall into a local minima, the compliance drops rapidly. Due to the trivial solution where $[K] = 0$, compliance drops to a very small number from the beginning (that is $\{u\}^T [K] \{u\} \approx 0$). Eventually this becomes ill-conditioned, and ultimately halts the loop once it becomes singular.

The total runtime for all options is in Figure (7.11(a)) and the speedup in Figure (7.11(b)). Again, a similar behavior is obtained when the GPU chain with CPU solver are selected, with a speedup close to 1.1 (faster than the CPU), and the compute chains that make use of the GPU solver are slower than the rest. The difference being that the speedup of these are closer to 0.20 instead of 0.25 as before.

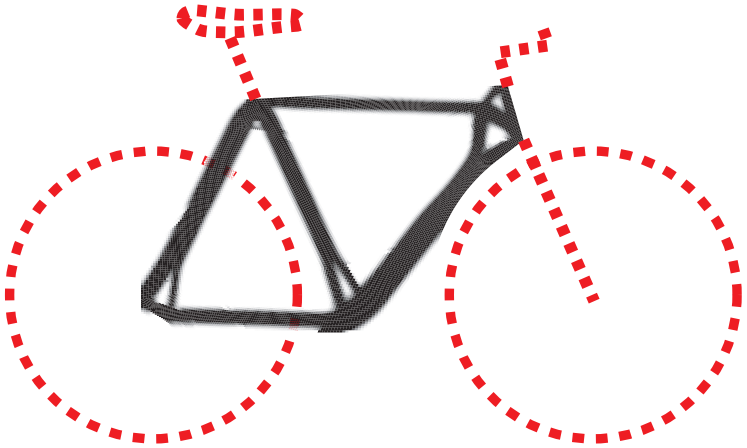


Figure 7.4: Resulting topology for the bike frame problem with the remaining components traced.

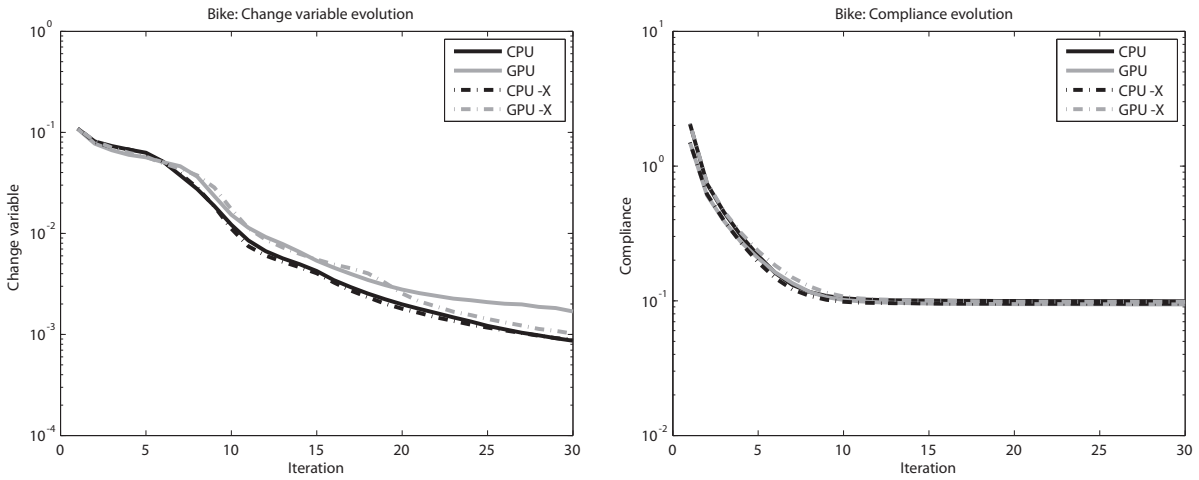
7.3 MBB Beam With Holes

The MBB beam with holes problem is a variation of the typical MBB beam, that includes two circular holes as a restriction in the design domain (only one hole for the half domain). The domain, loading and boundary conditions are in Figure (7.12). The mesh has 55200 elements, 55900 nodes and a bandwidth of 690. Because the bandwidth is bigger than 513, the GPU solver cannot be used to solve this problem, reducing the possible compute chains to only two (CPU and GPU -X). The mesh can be seen in Figure (7.13).

The topologies were obtained for a volume fraction $volfrac = 0.3$, filter radius $R = 0.02$ and 30 iterations (same as the MBB beam problem). Results for all compute chains (only two in this problem) are available in Figure (7.14). It is of great interest to note that the hybrid GPU chain ($GPU - X$) resulted in a design with a larger number of fine topologies, despite having the same filter parameters of the CPU chain (a generally desired thing). The CPU chain design resembles a truss, while the hybrid GPU chain achieves a more continuous design, specially evident from the more curved arch in the final design (or *extrados* in arch bridges jargon). Nevertheless, the CPU final design is more crisp when compared to the hybrid GPU one.

The compliance and change variable for both problems decrease monotonically (Figure (7.15)). This indicates that both compute chains are stable and converging properly, even though the topologies from each one are quite different. The hybrid GPU chain converges to a lower compliance than the CPU chain, that matches the previous discussion regarding the better design achieved by the GPU.

The total runtime for all compute chains is in Figure (7.16(a)) and the speedup in Figure (7.16(b)). This example follows the same behavior seen in the previous examples, where the GPU chain with CPU solver perform almost the same as the CPU chain. Nevertheless, in this case, the speedup is actually closer to 1.0



(a) Change variable

(b) Compliance

Figure 7.5: Evolution of the change variable and compliance for the bike frame problem for 30 iterations.

(no speedup at all).

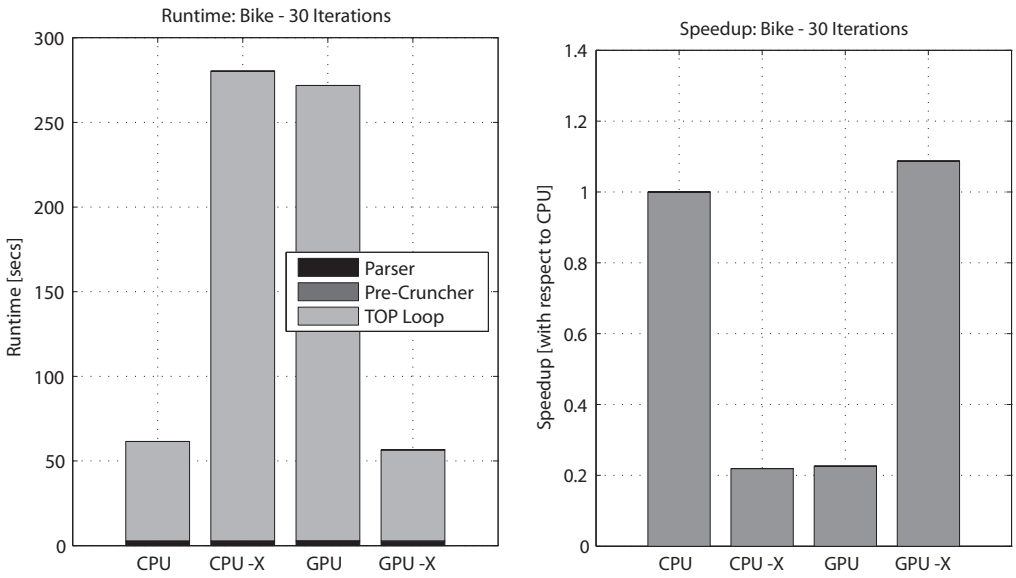
7.4 Hybrid GPU: TOP Algorithm Profiling

Currently, the fastest compute chain that makes use of the GPU is the hybrid GPU chain (CPU solver). For this specific case, the code is profiled to see where the time is spent within the topology optimization algorithm. The topology optimization loop will be sub-divided into 3 main parts: Solver, Memory transfer (GPU→CPU or CPU→GPU) and Others (Assembly, OC, Sensitivity, etc. . .).
The cases run are the same as before: The bike frame, the MBB beam and the MBB beam with holes. The result of this profiling is in Figure (7.17).

Results indicate that approximately 90% of the time is spent solving the system, and the memory transfers and rest of the functions account for the remaining 10%. Because we are using a banded storage, with bandwidth optimizers, the storage is very efficient and the memory transfers represent less than 3% of the total runtime of the TOP algorithm. The overall code can experience the most speedup, if optimizations or improvements get applied to the solver section of the code.

7.5 Remarks

The GPU effectively does the topology optimization algorithm slightly faster than the CPU, but in most cases the speedup is lost in the solver. For the case of the hybrid GPU chain, some of the speedup is lost in



(a) Code runtime

(b) Speedup

Figure 7.6: Benchmarks for the bike frame problem after 30 iterations.

the memory transfer.

The resulting topologies slightly differ depending on the compute chain that was selected. Nevertheless, the factor that has the biggest impact on the solution is the solver. When comparing the CPU and GPU chains, both using the same solver, the differences are very small. The solution from the GPU is less crisp and clean when compared to the CPU for very large problems (like for the MBB beam with holes), because of the slightly lower accuracy in the computations and precision of the GPU compared to the CPU. In the current GPU solver, the BaCh solver introduces more errors in the solution compared to its CPU counterpart in

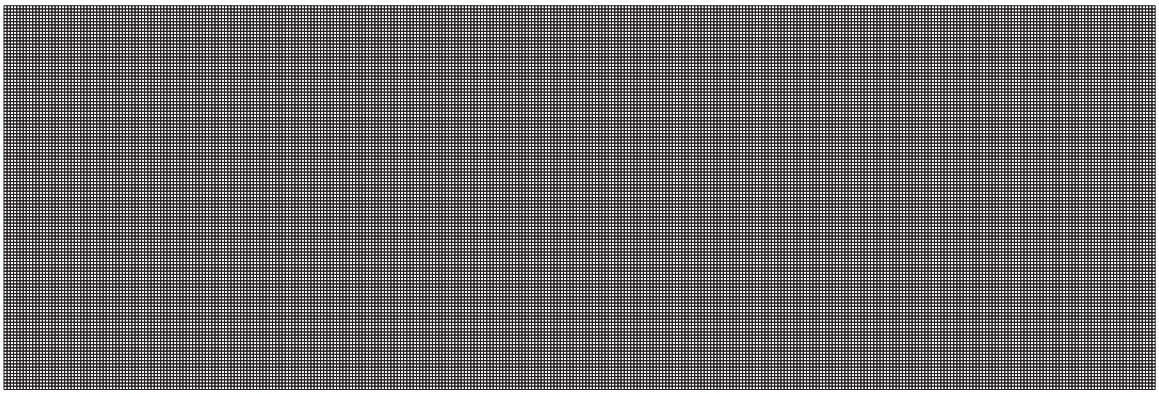
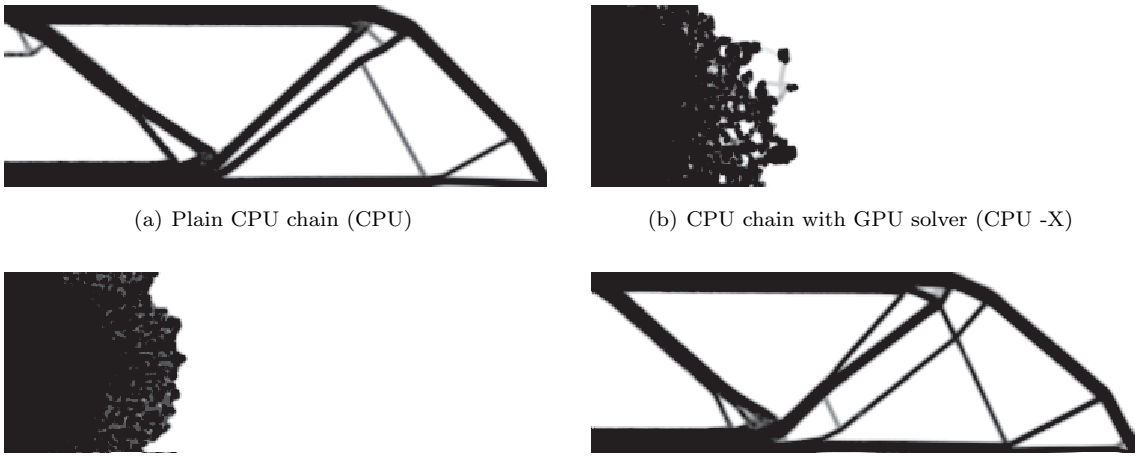


Figure 7.7: MBB beam mesh (43200 elements, 46381 nodes).



(a) Plain CPU chain (CPU)

(b) CPU chain with GPU solver (CPU -X)

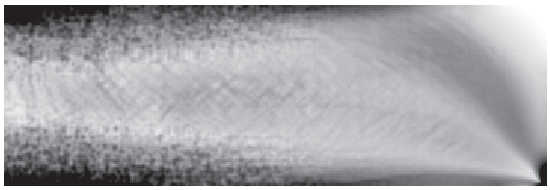
(c) Plain GPU chain (GPU)

(d) GPU chain with CPU solver (GPU -X)

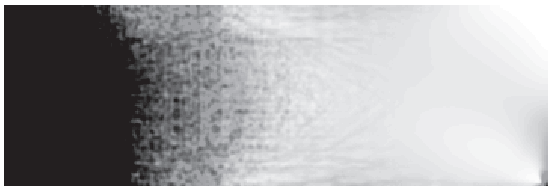
Figure 7.8: MBB beam results for all 4 compute chains after 30 iterations each.

LAPACK. For the case of the MBB beam problem, an invalid solution is obtained as the algorithm converges to an undesirable topology (the compliance and change variables still behave normally).

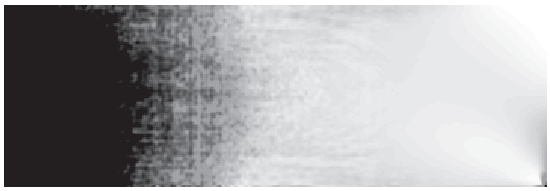
The GPU chain should be able to outperform the CPU if a fast enough solver is implemented in the GPU. Explicit and iterative solvers are better suited for this type of architectures, and may possibly outperform the one available in LAPACK.



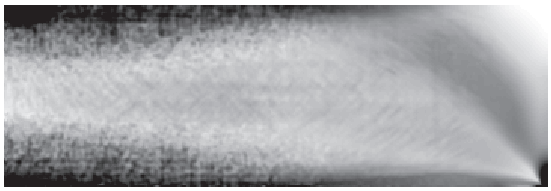
(a) Plain CPU chain (CPU)



(b) CPU chain with GPU solver (CPU -X)

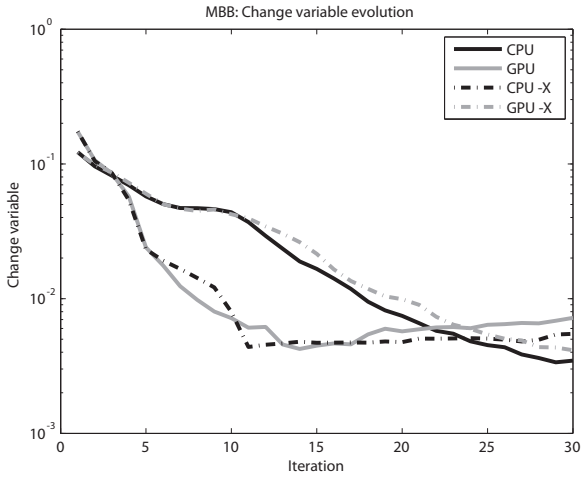


(c) Plain GPU chain (GPU)

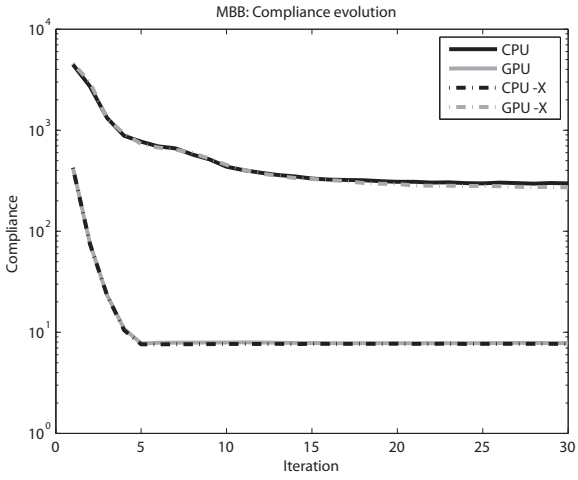


(d) GPU chain with CPU solver (GPU -X)

Figure 7.9: MBB beam results for all 4 compute chains after 2 iterations each.

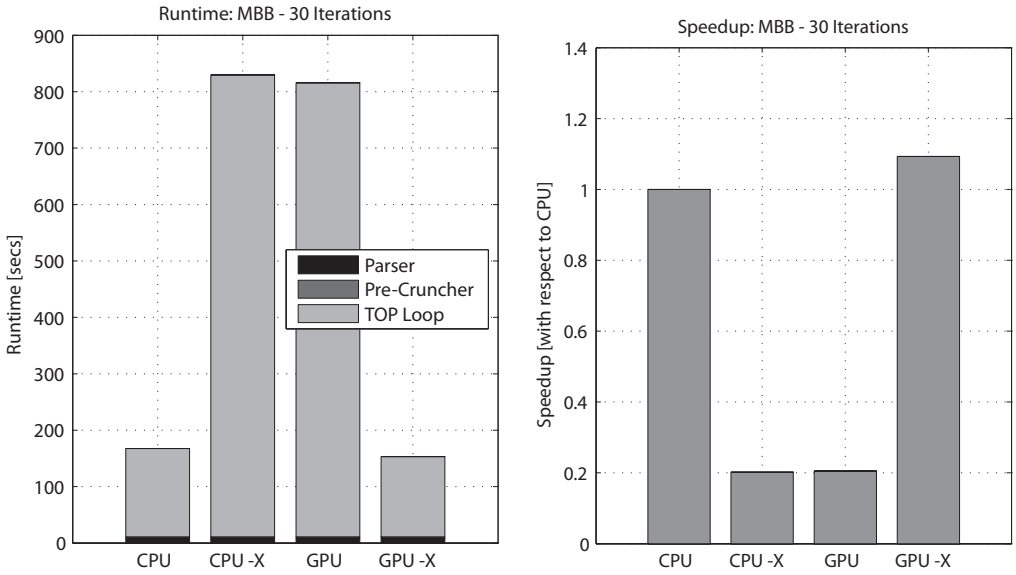


(a) Change variable



(b) Compliance

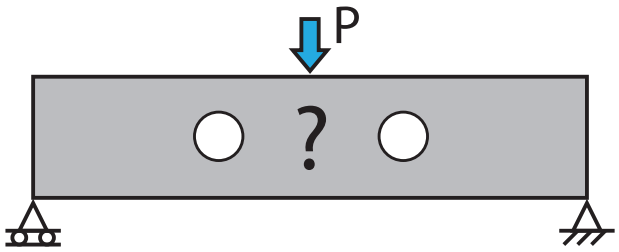
Figure 7.10: Evolution of the change variable and compliance for the MBB beam problem for 30 iterations.



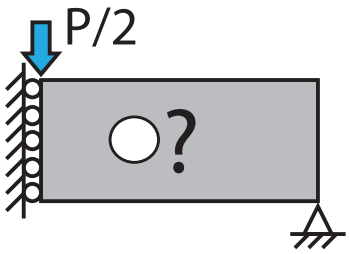
(a) Code runtime

(b) Speedup

Figure 7.11: Benchmarks for the MBB beam problem after 30 iterations.



(a) MBB beam with holes problem.



(b) Symmetric half of the MBB beam with holes.

Figure 7.12: MBB beam with holes problem.

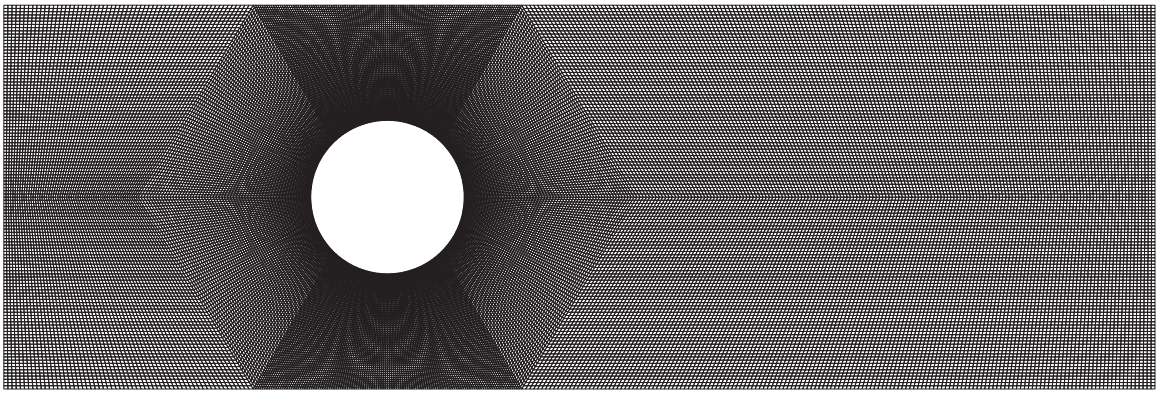
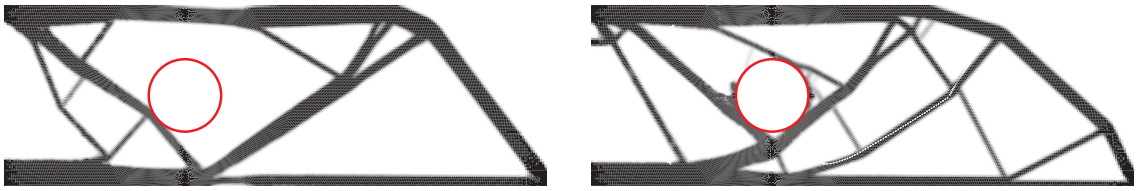


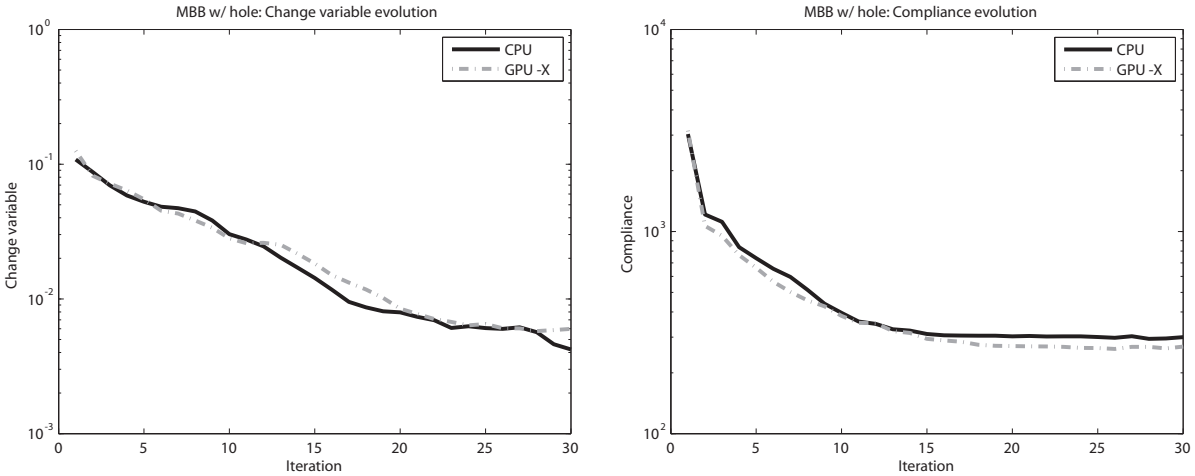
Figure 7.13: MBB beam with holes mesh (55200 elements, 55900 nodes).



(a) Plain CPU chain (CPU)

(b) GPU chain with CPU solver (GPU -X)

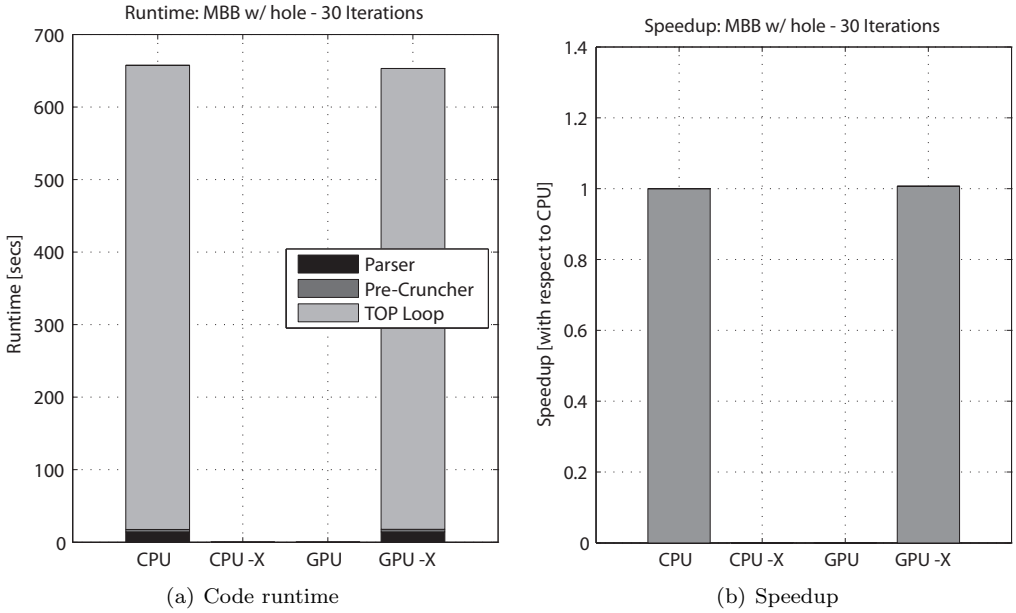
Figure 7.14: MBB beam with holes results for both compute chains after 30 iterations each.



(a) Change variable

(b) Compliance

Figure 7.15: Evolution of the change variable and compliance for the MBB beam with holes problem for 30 iterations.



(a) Code runtime

(b) Speedup

Figure 7.16: Benchmarks for the MBB beam with holes problem after 30 iterations.

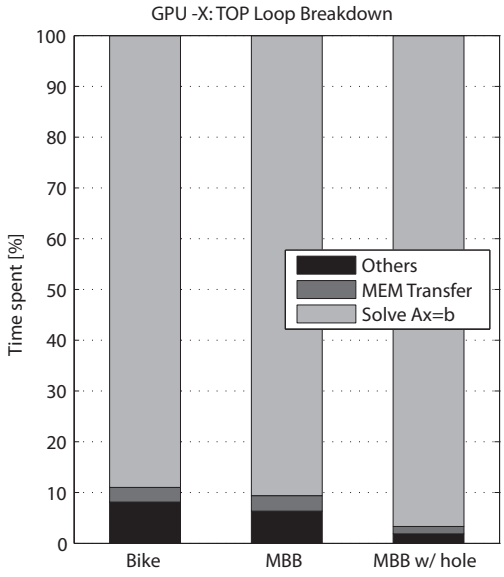


Figure 7.17: Hybrid GPU (CPU -X) code profile for the topology optimization algorithm.

Chapter 8

Conclusions and Recommendations for Future Work

Topology optimization in GPUs for unstructured meshes is indeed plausible and potentially faster than the CPU. However the speedup gained from assembling, computing the sensitivities, updating the material and other steps is lost in the solver (slower than its CPU counterpart). The solver controls the overall speedup since it takes up for approximately 90% of the topology optimization iterations runtime.

The GPU code operates under single precision, and only supports normalized floating points in current NVIDIA architectures (based on the G80 core). This results in differences in the design when compared to the same single precision but de-normalized floating points in the CPU. And in some rare cases, it even fails to converge.

Because the mesh is unstructured, a significant amount of work must be carried out before the topology optimization loops commence. To prevent a possible race condition, a simple and fast graph coloring scheme is applied to the mesh. The graph coloring problem is NP-hard, and an optimal solution is hard or impossible to obtain in a reasonable time. Nevertheless, the problem is somewhat relaxed by the maximum number of threads, resulting in an efficient coloring regardless of the simplicity of the coloring algorithm. Filtering for unstructured meshes also does not scale well. Because of that, the filtering is precomputed and stored so that minimal work is done every time the filter is applied within the iterations. The current state of the code makes no difference between structured or unstructured meshes. For the case of a structured mesh, for the coloring only 4 colors are required, and the filter search is simple. This would result in a very fast and efficient code [39]

The use of unstructured meshes allows the user to define complex domains, loads and restrictions. The advantage is clear over a structured mesh, specially for real life applications where the domain is rarely orthogonal, and borders do not necessarily follow straight lines. Structured meshes have the advantage of

having repetitive and simple local stiffness matrix, the race condition problem can also be easily solved, and the resulting matrix has a structured profile that can be efficiently solved. The TOP algorithm losses flexibility with a structured mesh.

The algorithms, ideas and code presented here are all applicable for the 3D case. Extension of the current code to 3D is relatively simple, and many functions and kernels would remain almost untouched.

8.1 Future Work

The code could be further improved in many aspects, but the changes that look the most promising to the authors are:

- Full support for 3D problem parsing from ABAQUS files, and extension of the current code to support 3D problems.
- Support for Continuous Approximation of Material Distribution (CAMD) approach [29].
- Faster and better coloring algorithms or race-condition-free assembly procedures [8].
- Add a flag to the code that indicates if the mesh is structured, and act accordingly [39].
- The current implementation of the BaCh solver is rather naïve, and code refinement may result in higher performance. Features such as the ability to handle matrices with bandwidths bigger than 513, the support for multiple column vector on the right-hand-side and support for upper triangular banded notation can also be added.
- Experiment with iterative solvers, other GPU solvers [45], and/or the soon to be released full implementation of BLAS in CUDA, named CUBLAS (currently in beta stage within CUDA v3.1-beta at the time of this writing).
- As the GPU architectures evolve, the limits on number of threads and memories should increase, resulting in faster or more powerful code.

In addition, there are algorithm changes in the way the code currently works that could also potentially make things faster. Some of these are:

- Asynchronous computing using the CPU and GPU in parallel.
- Effective usage of more than just one GPU.

- Double precision and de-normalized floating points in all computations without performance hit (future NVIDIA architectures).
- New CUDA features such as *Zero-copy memory access* [64] allows the GPU to directly access the CPU memory possibly improving performance or increasing the maximum problem size.
- Concurrent kernel execution.

8.2 Final Remarks

There is quite a lot of room for improvements, ideas and combinations to be explored. The current $1\times$ speedup may not seem appealing, but this code is the very first version and aims to prove the feasibility of topology optimization for unstructured meshes in the GPU. From this point onwards, any improvement should take the speedup above 1, resulting in a better-than-the-CPU code.

With the current computer architecture trend, with an increasing number of cores and threads while keeping the clock rate at its present value, massively parallel algorithms are on the rise, and are taking over sequential codes.

Future improvements in the GPU architectures are likely to eliminate most of the problems encountered here. This means that problems should almost always converge to the correct solution, and those obtained from the GPU and CPU should have no noticeable difference.

In conclusion, in order to effectively harness the compute capabilities of future computer architectures, the codes should be massively parallel and scalable. Parallel codes are here to stay, and represent the future of high performance and efficient computing.

References

Papers, Documents & Publications

[1] Ananiev S: *On equivalence between optimality criteria and projected gradient methods with application to topology optimization problem.* Multibody System Dynamics 13, No. 1, (2005), pp. 25-38

[2] Bell N, Garland M: *Efficient sparse matrix-vector multiplication on CUDA.* NVIDIA Technical Report, (2008)

[3] Bendsøe MP: *Optimal shape design as a material distribution problem.* Structural Optimization 1, (1989), pp. 193-202

[4] Bendsøe MP, Kikuchi N: *Generating optimal topologies in structural design using a homogenization method.* Computer Methods in Applied Mechanics and Engineering 71, No. 2, (1988), pp. 197-224

[5] Bendsøe MP, Sigmund O: *Material interpolation schemes in topology optimization.* Archives of Applied Mechanics 69, No. 9-10, (1999), pp. 635-654

[6] Bruns TE: *A reevaluation of the SIMP method with filtering and an alternative formulation for solid-void topology optimization.* Structural and Multidisciplinary Optimization 30, (2005), pp. 428-436

[7] Carvalho RF, Martins CAPS, Batalha RMS, Camargos AFP: *3D parallel conjugate gradient solver optimized for GPUs.* 14^{th} Biennial IEEE Conference on Electromagnetic Field Computation (CEFC), (2010)

[8] Cecka C, Lew AJ, Darve E: *Assembly of finite element methods on graphics processors.* International Journal for Numerical Methods in Engineering, (2010)

[9] Clough RW: *Original formulation of the finite element method.* Finite Elements in Analysis and Design 7, No. 2, (1990), pp. 89-101

[10] Cuthill E, McKee J: *Reducing the bandwidth of sparse symmetric matrices.* Proceedings of the 24^{th} National Conference of the ACM, (1969)

[11] Dziekonski A, Lamecki A, Mrozowski M: *Jacobi and Gauss-Seidel preconditioned complex conjugate gradient method with GPU acceleration for finite element method.* Proceedings of the 40^{th} European Microwave Conference, (2010), pp. 1305-1308

[12] Ergatoudis I, Irons BM, Zienkiewicz OC: *Curved, isoparametric, "quadrilateral" elements for finite element analysis.* International Journal of Solids Structures 4, (1968), pp. 31-42

[13] Felippa, CA: *A historical outline of matrix structural analysis: a play in three acts.* Computers & Structures 79, No. 14, (2001), pp. 1313-1324

[14] Gebremedhin AH, Manne F, Pothen A: *What color is your Jacobian? Graph coloring for computing derivatives.* SIAM Review 47, No. 4, (2005), pp. 629-705

[15] Gibbs NE, Poole WG, Stockmeyer PK: *An algorithm for reducing the bandwidth and profile of a sparse matrix.* SIAM Journal of Numerical Analysis 13, No. 2, (1976), pp. 236-250

[16] Gibbs NE, Poole WG, Stockmeyer PK: *A comparison of several bandwidth and profile reduction algorithms.* ACM Transactions on Mathematical Software 2, No. 4, (1976), pp. 322-330

[17] Gödel N, Schomann S, Warburton T, Clemens M: *GPU accelerated Adams-Bashforth multirate discontinuous Galerkin FEM simulation of high-frequency electromagnetic fields.* IEEE Transactions on Magnetics 46, No. 8, (2010), pp. 2735-2738

[18] Golub GH, Welsch JH: *Calculation of Gauss quadrature rules.* Mathematics of Computation 23, (1969), pp. 221-230

[19] Halfhill TR: *Parallel processing with CUDA.* Microprocessor Report, (2008)

[20] Harris MJ, Coombe G, Scheuermann T, Lastra A: *Physically-based visual simulation on graphics hardware.* Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, HWWS '02, (2002)

[21] Irons BM: *Engineering application of numerical integration in stiffness methods.* AIAA Journal 4, No. 11, (1966), pp. 2035-2037

[22] Kakay A, Westphal E, Hertel R: *Speedup of FEM micromagnetic simulations with graphical processing units.* IEEE Transactions of Magnetics 46, No. 6, (2010), pp. 2303-2306

[23] Kazimi SMA: *Solution of plane-stress problems by grid analysis.* Building Science 1, No. 4, (1966), pp. 277-288

[24] Kučera L: *The greedy coloring is a bad probabilistic algorithm*
Journal of Algorithms 12, No. 4, (1991), pp. 674-684

[25] Lawler EL: *A note on the complexity of the chromatic number problem.* Information Processing Letters 5, No. 3, (1976), pp. 66-67

[26] Liu WH, Sherman A: *Comparative analysis of the Cuthill-McKee and the reverse Cuthill-McKee ordering algorithms for sparse matrices.* SIAM Journal of Numerical Analysis 13, No. 2, (1976), pp. 198-213

[27] Liu Y, Jiao S, Wu W, De S: *GPU accelerated fast FEM deformation simulation.* IEEE Asia Pacific Conference on Circuits and Systems (APCCAS), (2008)

[28] Mahdavi A, Balaji R, Frecker M, Mockensturm EM: *Topology optimization of 2D continua for minimum compliance using parallel computing.* Structural and Multidisciplinary Optimization 32, No. 2, (2006), pp. 121-132

[29] Matsui K, Terada K: *Continuous approximation of material distribution for topology optimization.* International Journal for Numerical Methods in Engineering, Vol. 59, No. 14, (2004), pp. 1925-1944

[30] Michell AGM: *The limits of economy of material in frame-structures.* Philosophical Magazine - Series 6, Vol. 8, No. 47, (1904)

[31] Moore GE: *Cramming more components onto integrated circuits.* Electronics 38, No. 8, (1965), pp. 114-117

[32] Nguyen TH, Paulino GH, Song J, Le CH: *A computational paradigm for multiresolution topology optimization (MTOP).* Structural and Multidisciplinary Optimization 41, No. 4, (2010), pp. 525-539

[33] NVIDIA Corporation: *Fermi compute architecture white paper*, (2009), http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[34] Oliker L, Biswas R: *Parallelization of a dynamic unstructured algorithm using three leading programming paradigms.* IEEE Transactions on Parallel and Distributed Systems 11, No. 9, (2000), pp. 931-940

[35] Owens JD, Luebke D, Govindaraju N, Harris M, Krüger J, Lefohn AE, Purcell TJ: *A survey of general-purpose computation on graphics hardware.* Computer Graphics Forum 26, No. 1, (2007), pp. 80-113

[36] Paulino GH, Menezes IFM, Gattass M, Mukherjee S: *Node and element resequencing using the Laplacian of a finite element graph. Part I: General concepts and algorithm.* International Journal for Numerical Methods in Engineering 37, No. 9, (1994), pp. 1511-1530

[37] Paulino GH, Menezes IFM, Gattass M, Mukherjee S: *Node and element resequencing using the Laplacian of a finite element graph. Part II: Implementation and numerical results.* International Journal for Numerical Methods in Engineering 37, No. 9, (1994), pp. 1531-1555

[38] Samuelsson A, Zienkiewicz, OC: *History of the stiffness method.* International Journal of Numerical Methods in Engineering 67, No. 2, (2006), pp. 149-157

[39] Schmidt S, Schulz V: *A 2589 line topology optimization code written for the graphics card.* Technical report Preprint SPP1253-068, (2009)

[40] Seiler L, Carmean D, Sprangle E, Forsyth T, Abrash M, Dubey P, Junkins S, Lake A, Sugerman J, Cavin R, Espasa R, Grochowski E, Juan T, Hanrahan P: *Larrabee: A many-core x86 architecture for visual computing.* ACM Transactions on Graphics 27, No. 3, (2008), Article 18

[41] Shinn AF: *Implementation issues for CFD algorithms on graphics processing units.* Guest Lecture for ECE498AL, University of Illinois at Urbana-Champaign, (2009)

[42] Sigmund O: *A 99 line topology optimization code written in Matlab.* Structural and Multidisciplinary Optimization 21, No. 2, (2001), pp. 120-127

[43] Sutter H: *The free lunch is over: A fundamental turn toward concurrency in software.* Dr. Dobb's Journal 30, No. 3, (2005)

[44] Svanberg K: *Method of moving asymptotes - a new method for structural optimization.* International Journal of Numerical Methods in Engineering, Vol. 24, No. 2, (1987), pp. 359-373

[45] Tomov S, Nath R, Ltaief H, Dongarra J: *Dense linear algebra solvers for multicore with GPU accelerators.* IEEE International Symposium on Parallel & Distributed Processing (IPDPSW), (2010)

[46] Trendall C, Stewart AJ: *General calculations using graphics hardware, with application to interactive caustics.* Eurographics Workshop on Rendering, (2000)

[47] Turner MJ: *The direct stiffness method of structural analysis.* AGARD Meeting, Aachen - Germany, (1959)

[48] Vázquez F, Ortega G, Fernández JJ, Garzón EM: *Improving the performance of the sparse matrix vector product with GPUs.* Proceedings of the 10^{th} IEEE International Conference on Computer and Information Technology (CIT), (2010), pp. 1146-1151

[49] Vemaganti K, Lawrence WE: *Parallel methods for optimality criteria-based topology optimization.* Computer Methods in Applied Mechanics and Engineering 194, No. 34-35, (2005), pp. 3637-3667

[50] Volkov V, Demmel JW: *Benchmarking GPUs to tune dense linear algebra.* Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, (2008), pp. 1-11

Books & eBooks

[51] Adeli H, Soegiarso R: *High-performance computing in structural engineering*
CRC, First Edition, (1998), Chapter 5.4

[52] Barker VA, Blackford LS, Dongarra J, Du Croz J, Hammarling S, Marinova M, Wasniewski J, Yalamov P: *LAPACK95 Users' Guide (Software, Environments and Tools)*
SIAM, First Edition, (2001)

[53] Belytschko T, Liu WK, Moran B: *Nonlinear finite elements for continua and structures*
Wiley, First Edition, (2000)

[54] Bendsøe MP, Sigmund O: *Topology optimization*
Springer, Second Edition (2002)

[55] Cook RD, Malkus DS, Plesha ME, Witt RJ: *Concepts and applications of finite element analysis*
Wiley, Fourth Edition, (2001)

[56] Foster I: *Designing and building parallel programs: Concepts and tools for parallel software engineering*
Addison Wesley, (1995)

[57] Haftka RT, Gürdal Z: *Elements of structural optimization (Solid mechanics and its applications)*
Springer, Third Revision and expanded Edition (1991)

[58] Heath MT: *Scientific computing - An introductory survey*
McGraw-Hill, Second Edition, (2002)

[59] Hemp WS: *Optimum structures*
Clarendon Press, First Edition, (1973)

[60] Hughes TJR: *The finite element method: Linear static and dynamic finite element analysis*
Dover Publications, (2000)

[61] Kirk DB, Hwu WW: *Textbook for: ECE498AL - Programming massively parallel processors*
University of Illinois at Urbana-Champaign, (2009)

[62] Kirk DB, Hwu WW: *Programming massively parallel processors: A hands-on approach*
Morgan Kaufmann, First Edition, (2010)

[63] Mattson TG, Sanders BA, Massingill BL: *Patterns for parallel programming*
Addison-Wesley Professional, First Edition, (2004)

[64] NVIDIA Corporation: *CUDA C programming - Best practices guide*, (2009)
http://www.nvidia.com/cuda/

[65] NVIDIA Corporation: *CUDA programming guide*, (2009)
http://www.nvidia.com/cuda/

[66] Peressini AL, Sullivan FE, Uhl JJ Jr: *The mathematics of nonlinear programming*
Springer, (1988)

[67] Press WH, Teukolsky SA, Vetterling WT, Flannery BP: *Numerical recipes - The art of scientific computing*
Cambridge University Press, Third Edition, (2007)

[68] Rozvany GIN: *Topology Optimization in structural mechanics*
Springer, First Edition (2003)

[69] Strang WG, Fix GJ: *An analysis of the finite element method*
Wellesley-Cambridge, Second Edition (2008)

[70] Zienkiewicz OC, Taylor RL, Zhu JZ: *The finite element method*
Butterworth-Heinemann, Sixth Edition, (2005)

Software & Others

[71] Advanced Micro Devices, Inc. (AMD): *ACML - AMD Core Math Library: Version 4.3.0, (2009)*
http://developer.amd.com/cpu/Libraries/acml/Pages/default.aspx

[72] Altair Engineering, Inc.: *Altair OptiStruct*
http://www.altairhyperworks.com/

[73] ANSYS, Inc.: *ANSYS*
http://www.ansys.com/

[74] Autodesk®: *Algor® Simulation*
http://www.algor.com/

[75] Computers & Structures, Inc.: *SAP2000*
http://www.csiberkeley.com/

[76] CULA Tools: *GPU Accelerated LAPACK*
http://www.culatools.com/

[77] FIGES A.S.: *TOSCA®*
http://www.figes.com.tr/

[78] *LAPACK - Linear Algebra Package: Version 3.2.1, (2009)*
http://www.netlib.org/lapack/

[79] MSC.Software: *Patran*
http://www.mscsoftware.com/

[80] NVIDIA Corporation: *CUDA: Compute Unified Device Architecture*
http://www.nvidia.com/cuda/

[81] SIMULIA (Dassault Systmes): *Abaqus FEA*
http://www.simulia.com/

[82] *STL - Standard Template Library (C++ Standard Library)*
http://en.wikipedia.org/wiki/Standard_Template_Library

[83] Strand7 Pty Ltd.: *Strand7*
http://www.strand7.com/

[84] Topologica Solutions, Inc.: *CATOPO®*
http://www.topologicasolutions.com/

[85] Panagiotis Michalatos and Sawako Kaijima: *TopoStruct: Topology Optimization Software*
http://www.sawapan.eu/

[86] Vanderplaats R&D, Inc.: *GENESIS Structural Analysis and Optimization Software*
http://www.vrand.com/

[87] Cannondale Bicycle Corporation: *Cannondale Bicycles*
http://www.cannondale.com/